The Dissertation Committee for Zhengting He

certifies that this is the approved version of the following dissertation:

Towards Real-Time HW/SW Co-Simulation with Operating System Support

Committee:

Aloysius K. Mok, Supervisor

Vijay K. Garg, Supervisor

Baxter F. Womack

James C. Browne

Anirudh Devgan

Joydeep Ghosh

Towards Real-Time HW/SW Co-Simulation with Operating System Support

by

Zhengting He, B.S.E.E.; M.S.E.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2007

To my family

Acknowledgments

I would thank my parents Zhongqiu Wu and Deren Ding. Without their continuous encouragement, this dissertation would not have been possible. I would also like to express my gratitude to my wife Ling Wu, who constantly supports me throughout the ups and downs in my Ph.D. studies. It is so lucky for me to have them in my life. I would like to thank my baby boy Ray He who has supported his dad in his own special way, using his charming smile. Because of him, I have made up my mind to finish my dissertation so that one day he can be proud of his dad.

I would like to show special thanks to my supervisors, Prof. Aloysius K. Mok and Prof. Vijay K. Garg, not only for their excellent academic advice, but also for all kinds of support in my life. I joined the Real-Time System Lab in the University of Texas at Austin in 2001. I can still vividly remember the first time I walked into Prof. Mok's office and he introduced me the area of real-time system. Step by step, he has guided me to focus my research on HW/SW co-simulation with real-time operating system support which not only is a truly interesting and emerging research problem, but also matches my technical background perfectly. I want to thank Prof. Garg for allowing me continue my dissertation work meanwhile working in Texas Instruments Inc. from 2004. Without his trust and help, I would not have had the chance to finish my Ph.D. in the United States. Furthermore, their integrity and ethical behavior will greatly influence the rest of my life. I am blessed to have them as my Ph.D. supervisors.

I wish to thank my committee members, Prof. Baxter F. Womack, Prof. James C. Browne, Prof. Joydeep Ghosh and Dr. Anirudh Devgan, for their constructive feedback on my dissertation. I'm honored to have them on my committee.

Last but not least, I would like to thank all current and former members in the Real-Time System Group for the wonderful friendship, especially Xiang Feng, Deji Chen, Wing-Chi Pong, Jianping Song, Jianliang Yi and Weirong Wang for all the joys we had together.

Zhengting He

The University of Texas at Austin
May 2007

Towards Real-Time HW/SW Co-Simulation with Operating System Support

Publication No. _____

Zhengting He, Ph.D.
The University of Texas at Austin, 2007

Supervisors: Aloysius K. Mok and Vijay K. Garg

A trend in the consumer electronics market is the demand for new applications that have a lot of similarities to older applications but the new ones impose more challenging and special-purpose performance requirements. In the digital signal processing (DSP) industry, this clearly reflects a transition from the design regime of general DSP to the application-specific DSP. From the design perspective, it means that the DSP core remains unchanged but more and more hardware (HW) accelerators, DMAs and bus architectures need to be integrated into the chip. A key in effecting this transition is the engineering capability to make sure that the design specification "matches" the application before detailed design starts. Therefore, application software (SW) needs to be developed in parallel with HW to verify the design specification at the system level. Enabling development and simulation of SW before the actual HW is available also reduce the time-to-market period which is another important benefit.

HW/SW co-simulation for design specification refinement imposes many challenging requirements to the simulation platform. The simulation components (*simcoms*)

modeling the real HW (rhw) modules to be designed and the application SW need to be integrated to carry out the simulation at system level. Simulation result needs to be accurate. Simulation speed should allow fast design space exploration and ease debugging complex application SW. HW and SW problems should be isolated cleanly since HW and SW engineers often do not have enough expertise in one another's domains. The simulator should be cost-effective. These requirements often conflict with one another. For example, achieving high simulation accuracy typically requires the simulation to be carried out at low level, which implies that the simulation speed is slow. A simulator allowing integration of *simcom*s and application SW for simulation is very expensive and thus only very few engineers can use it. In many cases, *simcom*s and application SW are not constructed in the same programming language. Interfacing them is not a trivial problem and often impacts the simulation speed severely. Using a single simulator requires the engineers to understand both HW and SW details that violates the requirement of HW/SW problem isolation. The bottom line is that a single simulator is not possible to fulfill all these requirements at the same time.

This dissertation describes three simulation tools for different usages. The first one models and simulates the real-time operating system (RTOS) together with the application SW. It is motivated by the fact that with the appearance of high performance DSPs, more and more tasks will be implemented as SW on a single DSP managed by an RTOS. Selecting the "right" RTOS before the SW is developed is very important. The tool is implemented based on SystemC and is configurable to support modeling and timed simulation of most popular embedded RTOSes. Timing fidelity is achieved by using delay annotation. The OS timing information is derived from published benchmark data. Application timing information can be profiled or estimated from similar legacy applications. The optimized conservative approach is taken to synchronize *simcom*s. Compared to other research work, an important contribution of this tool is an online algorithm for predicting the timestamp of the

next event based on the realistic assumption that multiple tasks execute concurrently on a processor, managed by a static or dynamic priority driven scheduler. The simulation speed is more than 3 orders of magnitude faster than commercial instruction set simulator (ISS) with comparable accuracy. The tool is used to assist in generation of an initial design specification.

The second tool is a system dataflow simulator (SDFS) and is used by the HW engineers to refine the HW specifications. It models the application by a parameter-driven conditional dataflow graph (CDFG) at the transaction level and the HW by a configurable HW graph at the cycle-accurate level. SDFS takes the application CDFG and HW graph as the input and carries out the simulation to catch the detailed HW activities, i.e., bus arbitration. It only requires the HW engineers to understand the application at the CDFG level. To carry out the system simulation at such a low level, many commercial simulators need to couple an ISS for application SW with an RTL simulator for *simcom*s that are typically 6 orders of magnitude slower than the *rhw* speed. The simulation error of SDFS is within 5% in most cases and the worst case error is within 13%, which is comparable to the ISS+RTL approach. But the simulation speed is only 4 orders of magnitude slower than the *rhw* speed. Compared to other similar research work that also models the system at CDFG level, SDFS can achieve higher simulation accuracy because of the following advantages: 1) it does not need a fixed application trace as input and thus is flexible enough to cover many simulation scenarios; 2) it does not assume a fixed cost for each functional block and thus is able to estimate the system performance under actual execution conditions; and 3) it is able to model the pipelined architecture common in modern DSPs. The proposed simulator is cost-effective since it is implemented in the SystemC language and can be executed on most PCs and workstations.

The third tool is a real-time simulation platform (RTSP) implemented on legacy DSPs. To the best of our knowledge, this is the first simulator that truly enables the application SW to be developed in parallel with HW by offering the

same SW development environment as if the rhw was available. To simulate the behavior of a rhw module, a corresponding $simcom$ is constructed running on a legacy DSP. The success of this simulation strategy hinges on a novel way to apply the concept of Real-Time Virtual Machines to simulation. Each legacy DSP employs a two level scheduler to enforce that each $simcom$ carries out the simulation at a proportional speed $(1/\gamma)$ to the rhw, so that any job that would finish at time t on the rhw will finish no later than $\gamma \cdot t + \triangle$ where $\triangle$ is a constant bound. Such a feature eliminates expensive synchronization between the $simcom$s. RTSP is proven to perform simulations faithfully and also is shown experimentally to be effective for real industry applications. For a rhw whose timing behavior can be accurately modeled by the SW behavior model, the simulation error is shown to be $< 5\%$. For very complicated rhw whose timing cannot be accurately captured by the behavior model, the simulation accuracy was shown to be excellent for the average case. The simulation speed is quite fast. For the selected audio and video applications, simulation is only $10X$ and $30X$ slower than rhw execution. The RTSP platform is practically zero-cost since legacy EVM boards can be reused for the purpose of simulation.

RTSP and SDFS can be used to complement each other. RTSP carries out the simulation at a higher level than SDFS and usually cannot capture activities on buses at every cycle. The information collected from SDFS determines the appropriate rate settings for $simcom$s to compensate for the resource competition. RTSP allows SW engineers to optimize the algorithm and suggest improvements to HW architecture. Suggested changes are fed to SDFS for refining the design specification.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction to HW/SW Co-Design of DSP

In recent years, we have seen a steady growth in the sophistication of device technology and complexity in system integration. An increased number of devices in system naturally implies a higher degree of integration and more complex designs for achieving system performance. requirements. Traditionally when creating embedded systems, designers partition the hardware (HW) and software (SW) in the early design stage. HW and SW engineers design their respective components in isolation, and communication between the two groups is minimal. The drawbacks of such a sequential design process are evident: short of the best possible implementation, problems in HW and SW component integration result from the uncertainty in what system functionality will actually be implemented, entailing higher costs and long development cycles [83]. As a consequence, HW/SW co-design has gained considerable attention both in industry and in academia. The key difference between the co-design approach and the traditional approach is that the former fosters the design of HW and SW in parallel. It models and simulates a system at various abstract levels, exploring the design space and analyzing the tradeoffs to meet the

system objectives [83].

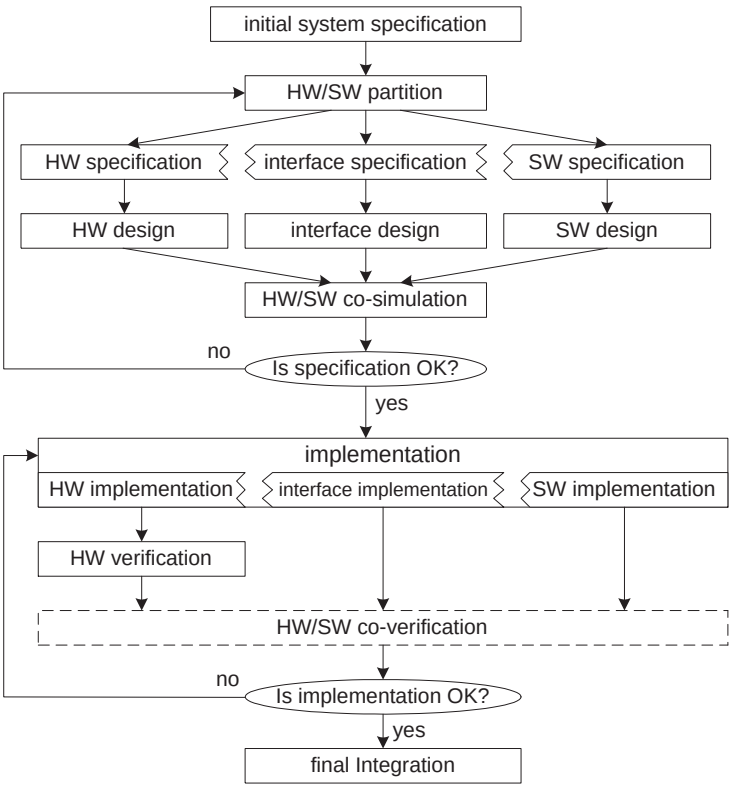

Figure 1.1: HW/SW Co-Design Process

Embedded systems cover a board area and different systems have very different characteristics. This dissertation focuses on an important application area - digital signal processor (DSP) based multimedia systems. A trend in the consumer electronics market is the demand for new applications that has a lot of similarities to older applications but the new ones impose more challenging and special-purpose performance requirements [98]. In the DSP industry, this clearly reflects a transition from the design regime of general DSP to the application-specific DSP. The typical design strategy is to leave the DSP core unchanged but redesign the on-chip HW accelerators, the DMAs and bus architectures for the new application [97].

A typical HW/SW co-design flow for a new DSP is depicted in Figure 1.1. It is divided into 2 phases, the **specification** phase and the **implementation** phase. The former phase starts with defining an initial system specification. Some decisions, such as whether to select a fixed point or a floating point core, which real-time operating system (RTOS) to deploy etc. are made and fixed at this stage. Other decisions like defining the memory architecture and HW accelerators etc. will be refined in later stages.

The HW/SW partitioning stage computes a good mapping of the system specification to a set of the real HW modules (*rhw*s) to be designed, i.e., DSP core, DMA, HW accelerators etc. Some functionalities are realized directly by the HW modules and an initial HW module level specification is generated. Other functionalities are implemented as SW being executed on the DSP core and similarly the initial SW module level specification is generated. The mapping is based on the estimation of the cost metrics on these modules. Typical HW cost metrics are execution time, chip area, power consumption and testability. SW cost metrics may include execution time and the amount of required program and data memory.

After the HW/SW partition is done, a simulation component (*simcom*) is constructed for each *rhw* at the desirable abstract level to model its functionality. An iterative simulation process is used to estimate the system performance until a satisfactory design specification is found. For examples, *simcom*s are constructed using the behavior model at the beginning and refined gradually to better reflect the timing characteristics. Typically the register-transfer-level (RTL) implementation for *rhw*s is not started in the specification phase. Application SW can be modeled and co-simulated with the *simcom*s in a unified environment to achieve fast simulation speed. The implementation can be started if legacy SW can be reused in which case an appropriate co-simulation platform is needed to integrate the SW with the *simcom*s. The interface is the SW driver to realize communication between the *rhw* and application SW. The communication behavior, i.e., synchronous

vs. asynchronous data transfer, also needs to be modeled and simulated together with all *simcom*s and application SW for determining the most appropriate driver structure. Implementation of the upper driver layers that are well isolated from the *rhw* can be started in the specification phase.

The other iterative process, the implementation phase, starts after the system specification has been generated. The implementation of the *rhw*s begins from the RTL level using VHDL or Verilog HDL language and is verified by the corresponding simulators. Then it is synthesized to gate level and verified on emulation HW such as FPGA. Eventually the physical layout is completed. Interface implementation can be made on the prototyping HW. Different platforms can be used to support HW/SW co-verification in this phase. Some platforms are SW based and some are implemented on emulation HW. Such simulators can achieve very accurate results but the simulation speed is slow. Long setup time, difficulty of debugging and extremely high cost forbid the development of any complex application SW [169]. In practice, HW/SW co-verification in the implementation phase is often optional. If any error that significantly affects system functionality or performance is found during the implementation phase, the design process has to switch back to the specification phase to re-generate a correct specification. Such a rollback can impose a huge cost and thus generating a correct specification that "matches" the targeted application is critical.

The rest of this chapter is organized as follows. Section 1.2 gives a literature survey of the HW/SW co-design area. Section 1.3 introduces some background knowledge on HW/SW co-simulation. Section 1.4 describes various simulation requirements in the specification phase and elucidates the problems to be solved. Section 1.5 summarizes the work done in this dissertation to solve these problems.

1.2 Survey of HW/SW Co-Design

The research on HW/SW co-design has been about 15 years. Co-Design is a design methodology supporting the concurrent development of HW and SW in order to achieve system functionality and performance goals [46]. It tries to increase the predictability of embedded system design by providing *analysis* methods that tell designers if a system meets its performance, power, and size goals and *synthesis* methods that let researchers and designers rapidly evaluate many potential design methodologies [168]. The co-design process for embedded systems includes many tasks such as specification, modeling, validation, and implementation. The field is fragmented because most efforts are applied to specific design problems [130]. The rest of this section gives an overview to HW/SW co-design from various perspectives. One thing to note is that co-simulation has always been considered as an important topic in the co-design area. Survey to research on co-simulation is given in section 1.3.

1.2.1 Complete Co-Design Environment

An important notion in co-design approaches is that there must not be a continuity problem. That is, the steps from model to the synthesis should all be in the design process [46]. Many tools have been proposed to offer a complete environment covering system specification, modeling, synthesis, validation and implementation.

- *Ptolemy II* [104] supports heterogeneous modeling, simulation and design of concurrent systems. It allows hierarchically combine a large variety of models of computation (MoCs) [68]. Most important MoCs are continuous time, discrete event, finite state machine (FSM) and periodic triggered actors [99]. The focus of Ptolemy II is on simulation. It is often used in combination with other tools, i.e. Polis.

- *Polis* [5] is a design environment for control-dominated embedded systems. It is based on co-design finite state machine (CFSM) and simulated in the discrete event domain of Ptolemy. The entire system is a group of communicating CFSMs plus an instruction set simulator (ISS) [139] [125]. The system specification language is Esterel [48], but a graphical specification can also be given. The SW is automatically generated. HW is synthesized as well, but off-the-shelf IP cannot be integrated [46]. Automatic partitioning is not supported in Polis.

  The Cadence Virtual Component Co-Design (VCC) [23] toolkit has been built on top of Polis. VCC allows better IP reuse than Polis.

- *COSYMA* [71][72] is an older design method. It covers the entire design flow from specification to synthesis. The target architecture is assumed to consist of a standard RISC processor, a fast RAM for program and data with single clock cycle access time and an automatically generated application specific co-processors. The modeling language is C_x which is a C-extension with support for parallel processes and timing constraints.

- *VULCAN* [85] is another older tool. The specification language used is called HardwareC. Although its syntax is C-like its semantics are that of a HW design language; thus rather low-level. Initially, a system will be specified as a complete HW solution. When the timing and resource constraints are specified, an iterative automatic partitioning approach moves suitable parts to SW running on a general purpose processor (GPP).

- *SpecC* [77] is a new language based on C. It includes a methodology for system design that allows a systematic design space exploration (DSE), called specify-explore-refine [76]. This methodology does not tend to support complex target platform [179]. The modeling approach used in SpecC is similar to ones used

in SW compilers, essentially a syntax graph.

- *STATEMATE* [92] is a set of tools, with a heavy graphical orientation, intended for the specification, analysis, design, and documentation of large and complex reactive systems. The modeling language is StateChart [126]. The main novelty of STATEMATE is in the fact that it "understands" the entire descriptions perfectly, to the point of being able to analyze them for crucial dynamic properties, to carry out rigorous executions and simulations of the described system, and to create running code automatically.

Not all approaches described above are "industry-ready". Ptolemy II and Polis are probably the most famous existing tools. However, both of them are only able to handle small systems in the control domain. Polis does not support IP reuse which prohibits it to be acceptable practically. Ptolemy II does not support sufficient SW and HW synthesis [46]. SpecC does take IP integration into account. However, it pays a price in a lower abstraction level than for example Polis and the partition is not automated [46]. The level of abstraction both in VULCAN and COSYMA is also low. The architecture assumption is too simple to keep pace with the complexity of current embedded systems. Research work on both of them has been stopped.

FPGAs have become an advantageous alternative to ASICs in low volume applications, due to its reduced cost [127]. Its design flow is similar to the ASIC design flow. Complete design environment offered by industry includes Xilinx [26], Altera [1] and Mentor Graphics [9].

1.2.2 Model of Computations and Languages

Modeling is the process of conceptualizing and refining the specifications, and producing a HW and SW model [130]. A model of computation (MoC) describes components in a system and how they communicate and execute [126].

The amount of MoCs used in the modeling language of a co-design method can be used to discriminate between two types of approaches: *homogeneous* modeling and *heterogeneous* modeling.

- *Homogeneous* modeling contains multiple MoCs in a single language. It is the most ambitious approach and has the advantage that the decision can be postponed what to implement in HW and what in SW. Many researchers doubt whether such an approach will work, because it is very difficult to combine various MoCs in a single language [54]. Examples of homogeneous methods are Polis [5], COSYMA [72] and SpecC [77]. The partition decision is supported by the homogeneous method itself. However, no real tools support this completely. An integral co-design approach is only possible for some specific algorithms or small set of applications [46].

- *Heterogeneous* modeling uses different languages to model HW and SW components. The choice between HW and SW implementation is made before the modeling phase starts, and each component is modeled in a language specially targeted towards that type of computation [130]. Simulation is used to validate the design and integration of the various languages (co-simulation). A typical example is Ptolemy II [104].

All co-design systems are based on an MoC or combine a few of them. For a specific MoC, the following three properties are important for system design.

- *Modeling of time* which can be continuous time, discrete time, partial ordered time (discrete event), or no explicit notion of time [54].

- *Orientation* which can be *state-oriented* reflecting the control sequence or *activity-oriented* reflecting the functionality. Examples of state-oriented MoCs are StateCharts [126] and CFSM [125]. Important activity-oriented MoCs are Kahn Process Network and Petri Nets [122].

- *Main application* which can either be dataflow oriented or control-flow oriented.

[122] proposed the tagged-signal model as a framework for comparing characteristics from different MoCs. It is clear that the choice of modeling language is fundamental to a co-design methodology. Extension of C/C++, i.e. SpecC and SystemC received a lot of interests recently. Although partition still cannot be automated, their benefits are: imposing a short learning curve to many engineers, allowing synthesis to many target architecture at RTL level, and supporting IP reuse.

1.2.3 HW/SW Partition

Partition is the task of allocating system functions to a set of HW and SW resources. In most practical cases, scheduling also needs to be considered as an integral part of the partition process. These two topics address *where* and *when* the system functions are implemented, respectively. The cost function of a partitioning problem needs to be evaluated using estimates of the resulting HW and SW. However, the abstraction level at which partitioning is carried out is so high that only rough estimates are available, which makes the partition problem hard [130].

Various formulations to the partition problem can be compared on the basis of the 1) *architectural assumptions*, 2) *partitioning goals*, and 3) *solution strategy* [130].

- *Architectural and application assumption*: [71] assumes that the co-processing HW is operated under direct control and in sequential with the GPP while [85] [55] assume that co-processing is concurrent with SW execution. [141] [166] [43] [138] [50] assume the architecture to be a set of symmetric processors. [57] [84] assume that the communication interface between the GPP and co-processor is memory mapped I/O while [171] assumes it is bus oriented. [136]

assumes that the communication operations between each pair of components are explicitly specified.

- *Partition objective*: Most algorithms try to maximize the overall speedup, i.e. [55] [161] [71] [106] [107]. [145] tries to minimize the overall cost. [62] provided a capacity constrained partition algorithm. [85] [109] are examples of performance constrained algorithms. [43] minimizes the synchronization overhead between HW and SW components.

- *Partition strategy*: [40] [136] [78] [144] provided mixed integer and integer programming algorithms. Many heuristic algorithms are published to partition a big set of functions to a complex architecture, i.e. [119] [164] [55]. [70] [71] are simulated annealing approaches. [63] used heuristic based task clustering, allocation, and scheduling approach. [67] is a genetic heuristic algorithm. In [45] [107], scheduling is applied before partition while [69] applies partition before scheduling.

A common problem in many existing partition tool is that some important parameters are rarely available, i.e. [33], or are heavily dependent on the designer experience on adequate number of reference implementations, i.e. [34]. [49] provided an affinity-driven partition tool in which a co-analysis step is performed to derive the characteristics for each function. The characteristics help measure if the function is suited for GPP or ASIC. The problems are two folds: 1) the affinity metrics are not accurate enough; and 2) system functionality needs to be realized as the input to the tool which is contradictory to the principal of HW/SW co-design.

The scheduling task can be decomposed to 3 subtasks: *HW scheduling*, *instruction scheduling* and *SW task scheduling*.

- *HW scheduling* is usually static. The scheduling algorithm can be integer linear program, i.e. [65] [78] [88] or list scheduling, i.e. [149]. Another approach is

called force-directed scheduling in which operations are scheduled into time slots, subject to time-window constraints induced by precedence and latency constraints, i.e. [140]. Online scheduling such as [118] is needed when some delay is unknown and synchronization is necessary.

- *Instruction scheduling* in compiler consists of front end to optimize operations on an intermediate form and a back end for code generation. When considering GPPs, instruction selection and register allocation are often achieved by dynamic programming algorithms which also generate order of instructions [30]. When considering retargetable compilers for application specific processors, the back end is more complex that instruction selection, register allocation and scheduling are tightly coupled with code generation [80].

- *Task scheduling* in a general OS primarily addresses increasing processor utilization and reducing response time. Scheduling algorithms are often rooted on simple procedures such as shortest job first or round-robin [44]. On the other hand, scheduling in RTOS primarily addresses the satisfaction of timing constraints. The algorithm can be static or dynamic in which the feasibility test is at run time [124] [147].

1.2.4 Synthesis

The partitioning and synthesis tasks are closely interrelated [130]. This section reviews some important work on synthesis other than partition.

Systems today are often modeled in C/C++ in the beginning and then translated by design engineers. Advantages in using the C/C++ language are numerous, i.e. large user base, modeling in a high level of abstraction, possibly automatic generation of SW and testbenches; fast execution etc. [37]. However, C/C++ does not allow explicitly declare parallel execution and arbitrary word size required by the actual HW [148].

To synthesize the HW using C/C++, two main techniques are deployed. The first one is called the *superset* approach which uses the C/C++ directly as the input to behavioral synthesis. Constructs and additional features are added to define parallelism, data-types, hierarchy, timing, and communication. These coding techniques may not be familiar and convenient to potential users, and hence are undesirable [148]. A famous example of the superset approach is SpecC [77]. The other approach is called *subset* approach which translates a subset of C/C++ into HDL which can eventually be synthesized using the already available commercial tools [4]. SystemC [81] belongs to the subset approach.

1.2.5 Platform Based Design for SoC

The industry has begun to embrace new design and reuse methodologies that are collectively referred to as system-on-chip (SoC) design to reduce the development cost and cycle. Due to the complexity of HW and SW, their reuse is often key to commercial profitability [130]. Typically a single but configurable architecture needs to serve many different customers in one market segment [152]. For these reasons, the concept of platform based design was introduced where new designs could be quickly created from an original platform to amortize costs over many design derivatives.

SoC is now a driver for the development and use of industry-wide standards. For example, standards of AMBA bus [14], OCP bus interface [10], IP exchange formats and documentation, IP protection and tracking [24], IEEE 1500 standard for test wrappers [17] have been developed and standardized.

The SoC design emphasis is on reusable IP design, integration, verification.

- *Reusable IP design*: Typically it may require five times as much work to generate a reusable IP block compared to a single-use block. The core creator should be responsible for delivering 1) the design-for-test (DFT) HW inside

the core; 2) the test patterns of the core; and 3) the validation of those test patterns. The validation work may take up to 50% of the total design time [110].

- *Integration*: The integration process involves connecting the IP blocks to the communication network, implementing DFT techniques and using methodologies to verify and validate the overall system-level design [146].

  A typical SoC today consists of many cores operating at different clock frequencies [152]. Each core can be viewed as a separate synchronous island, and the interfaces between these islands are provided by the chip's global interconnect. This approach is commonly referred to as globally asynchronous, locally synchronous (GALS) design [155] [134]. Synthesizing designs from multi-timed, transactional descriptions is an important topic of ongoing research. [154] [56] have exploited pipeline to implement a general set of interfaces between timing domains without fixed relationship. Optimized interfaces, i.e. [52] [129], can be implemented when special timing relationships are known.

- *Verification* of SoC is imposing much more challenges comparing to the traditional board level verification. The final test is to be applied at the input/output pins of the SoC. However, the core may be embedded deep into the SoC and thus its I/O pins may not be directly accessible from the external pins [152].

  While design sizes have grown exponentially over time in accordance with Moore's Law, theoretical verification complexity has been growing double-exponentially, because the number of states that must, in theory, be verified is exponential in the size of the design [8]. Most research activity, therefore, has focused on formal verification which provides a 100% proof that a design meets its specifications [61] without using test vectors. It is already indispensable industrial practice for RTL-to-gate equivalence checking [103] [169] and

microprocessor verification [42] [153].

The key research topics of formal verification are *compositional model checking* which decomposes the verification of an entire system composed of several blocks into multiple, smaller verification tasks on the individual blocks [60]; and *assume-guarantee reasoning* which emphasizes verification of cores under assumptions about the behavior of the rest of the system [142].

Since formal methods generally require a large memory space, semi-formal analysis is often chosen as a trade-off by mixing the formal techniques with the simulation-based approaches.. Typical example is assertion-based verification [27] [28] [39].

Testbench automation in simulation based verification approach is still important when the design size is very large. In many practical cases, it is much easier to create large amount of test vectors than to describe the complete specification for the design in a formal model [169].

IP-wise emulation using FPGAs has quite limited usage due to its high cost and debugging difficulty. Therefor, major industry and academia interest is how to get easier debugging environment with lower investment.

1.2.6 SW Toolkit Generation

The trend toward smaller mask-level geometries leads to higher integration and higher cost of fabrication, hence of amortizing HW design over large production volumes. This suggests the idea of using SW as means of differentiating products based on the same HW platform [130].

Rapid SW prototyping is often being overlooked in many existing HW/SW co-design methodologies. On one hand, it means that "slack" margins for HW, i.e CPU and memory resources, should not be overly restrictive to make the SW difficult design and implement [64]. One the other hand, it implies that there is a

critical need for support of an integrated SW development environment including compilers, simulators and debuggers [90].

The interest in the architectural description language (ADL) based design methodology for embedded SoC optimization and exploration has been tremendous. ADL is designed to specify architectural templates for SoC including components (processors, memories etc.), the interconnect, and the functionality of each component. The benefits of using the ADL not only include the ability to perform formal verification and consistency checking, but also to automatically generate the SW toolkit from a single specification. A typical SW toolkit include instruction level parallelizing (ILP) compilers, cycle-accurate ISSes, assemblers/disassemblers, profilers, debuggers etc.

ADLs can be categorized into 1) *behavior centric* ADLs, such as nML [93] and ISDL [86]; 2) *structure centric* ADLs such as MIMOLA [36] and COACH [31]; and *mixed level* ADLs such as LISA [181], EXPRESSION [89] and FlexWare [140].

A promising approach to automatic compiler generation is the "retargetable compile" approach. A compiler is classified as retargetable if it can be adapted to generate code for different target processors with significant reuse of the compiler source code. Recent approaches to retargetable compilers have focused on developing optimizations/transformations that are "retargetable" and capturing the machine specific information needed by such optimizations in the ADL [90]. Typical approaches to generate retargetable compilers include 1) *architecture-template-based*, i.e. [105] [13]; 2) *explicit-behavioral-information-based*, i.e.[91] [121]; and 3) *behavioral-information-extraction/generation-based*, i.e. [123] [89].

Simulation of the processor system can be performed at various abstraction levels. ISS is the highest level of abstraction. At lower-levels of abstraction are the cycle-accurate and phase-accurate simulation models [90]. Retargetable simulators can be categorized into 1) *interpretation based*, i.e. [111] [87]; 2) *compilation based*, i.e. [181]; and 3) *structure-centric ADL based*. Interpretation based simulators are

easy to implement and maintain by paying the price of slow speed (2K - 20K instructions/s). Compilation based simulators translate each target instruction directly to one or more host instructions at compile time and thus can be 3 orders of magnitude faster than interpretation based ones [180].

1.2.7 Open Problems

Researchers are still working on the following problems [168].

- Define and redefine MoCs for jointly describing HW and SW systems.
- System-level performance analysis is a complex problem that analysts must study under a variety of operating conditions suitable for various application types.
- Evaluate algorithms for DSE.
- Analyze new classes of architectures; develop new methods for performance analysis and code generation with the aim of making VLIW-based architectures more usable.
- Evaluate the effort of networks on chips (NoCs).
- Make the co-design tool friendly for SW development.

1.3 Introduction to HW/SW Co-Simulation

Verification of system functionality and timing is one of the most important and difficult aspects of the embedded system design. With the increasing complexity of modern embedded systems, formal method only has limited usage in verification of IP functionality. As a result, HW/SW co-simulation is the typical approach used for system level verifications nowadays. Recently the research on HW/SW co-simulation

mainly focus on four topics: 1) mixed-simulation of components in different abstract levels, 2) integration of multiple simulators implemented in different languages into the same environment, 3) simulation speedup by reducing synchronization overhead between *simcom*s, and 4) RTOS modeling.

1.3.1 Different Simulation Abstract in Level

From simulation abstract in level perspective, simulation can be roughly classified into the following four categories:

- **Gate level simulation**, also being called event driven simulation or phase-accurate simulation, is the most accurate since every active signal is calculated for every device during the clock cycle as it propagates. Each signal is simulated for its value and its time occurrence. It is excellent for timing analysis of HW circuit and verifying race conditions. Typically such simulators are implemented on emulation HW, i.e. FPGA or a quickturn machine [11]. All HW *simcom*s are synthesized to the emulation HW. The simulation speed of such HW based platform is typically $3-4$ orders of magnitude slower than the *rhw* speed, which is acceptable. However, the number of such multi-million machine is very limited. Transferring the input/output from/to the machine and programming it to enhance the debugging capability typically require additional HW support and can be very difficult. Therefore, such simulators are intended for HW verification at unit level but not for performance estimation at system level.

- **Cycle-accurate simulation** only calculates the state of the signals at clock edges and is often implemented by SW. Typical examples are RTL simulators coupled with ISSes. Compared to the gate level simulator base on emulation HW, this type of simulator is much slower but costs much less. Due to the SW implementation nature, virtually any signal can be captured and debugged. It

can be used for complex design and system level testing.

- **Transaction level simulation** uses SW function calls to model the communication between *simcom*s in a system. For example, a transaction level model (TLM) would represent a burst read or write transaction using a single function call, with an object representing the burst request and another object representing the burst response [160]. It is possible to create TLMs that are fully cycle-accurate, however in many cases it is applied to speed up simulation by foregoing full cycle accuracy [81].

- **Dataflow simulator** represents signals as stream of values without notion of time. Functional blocks are linked by signals and executed when signals present at the input. The scheduler in the simulator determines the order of block executions. It is a high level abstraction simulation used in the early design stage only for checking the correctness of the algorithms.

From the SW perspective, its access to the HW can be simulated at one of the following three abstract levels [175].

- *ISS* level in which the simulation cannot be started until all the SW code is ready to be compiled and linked.

- *Device driver level model* in which the target OS has been selected and the device driver functions access the abstract memory model when they are called by SW tasks or OS code.

- *OS level model* in which the target OS is yet to be selected or implemented. The OS is being modeled and only SW tasks are available.

In reality, many simulators need to support mixed-level HW and SW models for various needs, i.e. ISS for SW simulation + TLM for HW simulation. The mixed-level co-simulation problem is how to manage many different abstraction levels of

SW and HW models [175]. There are 12 total combinations of abstract HW and SW models in level and most existing solutions cover a subset of them. For example, in [176] the HW is TLM and the SW is the device driver level model. In [178], HW is TLM and SW is the OS level model.

Generally speaking, there are two types of approaches solving the mixed-level co-simulation problem. The first one is to support simulation of all the abstraction levels of the HW interface, i.e. [156]. Its advantages are two folds: 1) communication protocol being simulated at high level can be validated against the simulation results obtained at low levels; and 2) interconnecting *simcom*s with different abstraction levels of HW interface do not need an additional adaptation. However, such simulators are difficult to implement and maintain. As a result, a more popular approach is to design wrappers that adapt different abstraction levels, i.e. [54] [177] [41] [170]. [135] aims at automatically generating simulation wrappers while minimizing the number of required library components.

Raising the abstraction levels of simulation helps speed up simulation. However, the yield is typically lower than being expected. It is because the dominance in simulation runtime changes as we raise the abstraction level of SW and HW simulation. For ISS SW simulation + cycle-accurate HW simulation, the HW simulation dominates the total runtime and the speed can be as low as 1K cycles/s. By raising the abstract level for HW simulation to TLM level, SW simulation may dominate the runtime and the typically speed is about 100K cycles/s. More than 10M cycles/s speed can be obtained by raising the SW simulation to device driver or OS level and the HW simulation to TLM level [175].

1.3.2 Heterogeneous vs. Homogeneous Simulator

From the implementation language perspective, simulators can be classified as homogeneous and heterogeneous.

- **Heterogeneous** simulator provides interfaces between *simcom*s written in different languages so that they can be simulated together. For example, VHDL supports the link of procedures, written in C-code, to the simulator through its foreign language kernel [22]. Verilog supports invocation of C functions by a so-called programming language interface (PLI) [38]. Primarily there are two reasons making the simulation speed of a heterogeneous simulator prohibitively slow for fast DSE in the specification phase. Firstly, a heterogeneous simulator often requires the simulation to be carried at cycle-accurate level. Secondly, the communication cost between the *simcom*s written in different languages is expensive. [82] presents a C+VHDL simulator for DSP design which requires about 1.5×10^6 host cycles to simulate a target cycle.

- **Homogeneous** simulator intends to use a single language for system specification and even implementation. For example, SystemC [6] provides a subset of C++ semantics and libraries combined with a simulation engine to support co-simulation from dataflow to cycle-accurate level. Some commercial tools are also available to synthesize *simcom*s specified by SystemC into VHDL *simcom*s [4]. Similarly, the N2C package from CoWare [32] adds necessary clocking to C language for system modeling. Compared to the heterogeneous simulator, the homogeneous one is more flexible to model *simcom*s in various abstract levels and thus can achieve faster simulation speed. [98] presented a SystemC based simulator which models the application SW with a conditional dataflow graph and carries the simulation at cycle-accurate level. It requires about 4.4×10^4 host cycles to simulate a target cycle. The transaction level simulator in [96] only requires about 2 host cycles to simulate a target cycle.

  The interfacing issue still exists in the homogeneous environment when multiple simulators need to be integrated together even if they are written in the same language, i.e. ISS + SystemC. Various approaches have been published

to address this issue. [73] extends the SystemC kernel to integrate the ISS. Specifically, ports and process for ISS is added into the SystemC kernel and the scheduler is also modified. The drawback is that *simcom*s are not properly synchronized and thus timing fidelity cannot always be guaranteed. [102] also proposed an approach integrating ISS, focusing on the data type conversion. The abstract data types of the C++ environment are first mapped to a binary representation by a bitmapping layer. The resulting bit-streams are transferred to a protocol layer, cut into slices according to the respective data bus width and forwarded into the external simulator. The experiment shows that the simulator is cycle-accurate and achieves the simulation speed of 4.5×10^6 cycles/s. However, the HW being simulated is a single processor core so that the synchronization problem between *simcom*s does not exist.

The heterogeneous approach can naturally support reusing and integration of existing intellectual property (IP) blocks and thus is often applied in the implementation phase. The homogeneous approach is more applicable in the specification phase because it bridges the gap between HW and SW description languages [41] and makes co-simulation easier and more efficient. It is expected that these two approaches will coexist in the foreseeable future.

1.3.3 Synchronization Overhead between Simulation Components

Increased number of modules being integrated into a system implies that the simulator needs to make multiple *simcom*s progress concurrently and synchronize to each other. The **conservative** approach synchronizes *simcom*s at every clock step and thus the causal order of event processing will not be violated. Such an approach is only appropriate for simulation of a "busy" system at gate or cycle-accurate level when HW activities at almost each clock cycle need to be captured. Otherwise, the simulation speed can be severely impacted because of the unnecessary synchroniza-

tion overhead.

From the algorithm perspective, research on synchronization overhead reduction can be classified into two categories: **optimistic** approach and **optimized conservative** approach. A rich set of literature on optimistic simulation algorithms have been published among which *Time Warp* is the most well known [74]. In Time Warp, a causality error is detected whenever an event message is received that contains a timestamp smaller than that of the process's clock. The event causing rollback is called a straggler. Recovery is accomplished by undoing the effects of all events that have been processed prematurely. Rolling back the state is accomplished by periodically saving the process's state and restoring an old state on rollback. Depending on when an anti-message is sent, the cancellation mechanism can be aggressive or lazy. In the aggressive cancellation, whenever a process rolls back to time t, anti-messages are sent immediately for all positive messages sent after t. In lazy cancellation, processes do not immediately send an anti-message upon rollback. Instead, they wait to see if the re-execution of the computation regenerates the same message; if the same message is created, there is no need to send an anti-message. Depending on the application being simulated, lazy cancellation may improve or degrade performance. Critics argue that no proof yet exists that Time Warp is stable, although it is in most practical cases. [75] demonstrated that state saving overhead can seriously degrade performance of many Time Warp programs, even if the state vector is only a few thousand bytes. Another problem associated with the optimistic simulation algorithms is that they are hard to implement and thus usually not available in most commercial tools.

The optimized conservative approach tries to reduce synchronization frequency by predicting the synchronization points. The technique is also called lookahead which refers to the ability to predict what will happen, or equally important, what will not happen in the future, based on knowledge of the application and events that have already been processed [74]. There are two main techniques for synchro-

nization prediction. The first is compiler analysis of the SW being simulated [101] [108], i.e. analyzing assembly code and using dynamic runtime algorithm to handle branch and loops [58]. The speedup can be 6x to 40x compared to the conservative approach, depending on the application being simulated. The other one considers the execution semantics of the system being simulated and takes advantage of it [174] [159], of which the most famous technique is called *virtual synchronization* [170] [172] [113] [112]. The basic idea of the virtual synchronization technique is to separate global time management of simulation from local simulators utilizing algorithm model. When a local simulator produces output samples, the time differences between output samples are recorded from the simulator. The actual global time of output samples is computed by the simulation backplane [114]. The limitation is that static costs are assumed for local and shared memory accesses and no blocking is allowed for a *simcom*. [114] combines the trace-driven simulation with virtual synchronization to solve the static cost problem. It consists of two parts running concurrently so that required memory space is reduced. In the first part, it captures the execution traces from processing components ignoring the global time management. In the second part, it reconstructs the global time information for cycle-accurate simulation behavior using trace-driven co-simulation. The simulation speed is in the range of 420K - 1500K cycles/s. The main problem is that trace-driven simulation does not have the dynamic scheduling capability which violates the simulation fidelity. Moreover, deriving application trace is not a trivial task.

Some research work tries to reduce the cost per synchronization from the implementation perspective. [41] provided a SystemC based simulator in which the foreign ISS is integrated into the same SystemC engine with other *simcom*s so that expensive inter-process communication (IPC) is eliminated. Similarly, [174] eliminates IPC by integrating the VHDL based *simcom*s with the C based application SW using the VHDL-C interface. [101] [100] proposed the idea to dynamically al-

ter the level of communication details during co-simulation. The communication bandwidth is reduced at times when the details are not required. [113] [112] [173] indicated that the synchronization overhead is affected more by the number of messages exchanged than by the message size and thus the overhead can be reduced by grouping messages. For example, all signals associated with the same event can be grouped, and a sequence of events can be grouped into a higher level message. Experiment shows that simulation speedup due to message grouping ranges from 30% to 70%.

1.3.4 RTOS Modeling

With the increasing complexity of embedded devices, it is common to see in the modern embedded system several SW programs running concurrently on a processor managed by an RTOS. Embedded SW typically has real-time constraints to satisfy in addition to functionality requirements. It is important to be able to validate all these properties together as early as possible in the design cycle, and in the context of running the embedded SW on top of an RTOS. Research on RTOS modeling and simulation can be categorized into the following three directions:

- *System Call Translation* which re-directs the system call into the host OS. It is often used with an ISS to verify the SW functionality [150].
- *Native OS* which compiles the target OS code and executes it on the *simhw*, i.e. VxSim [25]. It does not support DSE.
- *Virtual OS* which simulates the functionality and timing of a real OS and is the most popular approach. Commercial tools include SoCOS [66] and CarbonKernel [3]. [79] presented an RTOS model based on SpecC, but the timing accuracy is not satisfactory. [131] presented an RTOS model on top of SystemC. Its focus is only modeling the scheduling algorithm and the process

communication mechanism. [29] [59] and [47] proposed the approach combining a SytemC based RTOS model with a bus functional model for HW. Timing is achieved by inserting delay annotations into the SW code; however, none of them shows how to easily get the delay information. The architectural assumption is a single-core-only system and the synchronization prediction ability is not available. [29] [59] did not say how to model the I/O behavior. [47] only models the blocking I/O for device drivers. For [29] [59], both simulation speeds are about 10x slower than the *rhw* speed with less than 10% error. [47] is 3 orders of magnitude faster than an ISS with less than 14% simulation error.

1.4 HW/SW Co-Simulation Requirements

1.4.1 Requirements of Generating an Initial Specification

To help generate an initial system specification, the following requirements need to be fulfilled by the simulator.

1. **RTOS modeling** needs to be supported by the simulator. The appearance of high performance DSPs has made it possible to implement more and more functionalities by SW. It is common to see in the modern embedded system several SW programs running concurrently on a processor managed by an RTOS. Embedded SW typically has real-time constraints to satisfy in addition to functionality requirements. It is important to validate these properties in the context of running the SW on top of an RTOS [96]. Traditionally, designers perform DSE by implementing the application SW to the target architecture which consumes a lot of time and is often error-prone [176]. RTOS modeling is a better technique that tries to model existing RTOSes within a single environment and captures the abstracted RTOS behaviors at the system level.

The model needs to be configurable and easy to use so that different candidates can be evaluated with little or no affection to the application SW.

2. **Speed** is one of the most critical requirements to enable fast DSE. It firstly implies that the target HW, application SW and RTOS should be modeled at the transaction level. Cycle-accurate or gate level simulation is not only too slow, but may also not available at this stage. Secondly, simulation synchronization overhead needs to be reduced. In general, the optimized conservative approach is considered to be more effective where the timestamps of the synchronization points can be approximately predicted with acceptable sacrifice to simulation accuracy. Because both the *simcom* and SW transaction level models can be described by the same language, i.e. SystemC or C, a homogeneous simulation environment is preferred due to its lower synchronization cost.

3. **Simulation accuracy** can be trade off for fast DSE in this stage. The most important criteria is to tell which RTOS or what kind of HW architecture is the most appropriate for the target application at a high level since these decisions have to be fixed. Exactly how much they are better comparing to other choices are less important since more accurate numbers can be obtained later using simulator with higher accuracy when the specification is refined.

1.4.2 Requirements of Specification Refinement

Specification refinement imposes more challenging requirements to the simulator.

1. **Simulation accuracy** is important to find the most appropriate system architecture. It is typically required to carry the simulation at the cycle-accurate level. Performance estimation not only needs to be conducted at unit level to tell how many cycles each functional block takes to execute on a rhw, more

importantly, system wide performance estimation is necessary to take into account the changes in the execution condition caused by races and different functional compositions [98].

2. **Speed** is still important not only for fast DSE, but also because it directly impacts the effort of debugging complex application SW [97].

3. **HW/SW problem isolation** is a practical concern since modern embedded systems have been complex enough that HW and SW engineers often do not enough expertise knowledge in each other's domain. It is desirable that changes made in one domain should have little or no impact to the other domain.

4. **SW development friendly** is another practical concern motivated from the fact that the application nowadays often needs to be developed by many SW engineers in parallel. Before the *rhw* is available, a low cost simulation platform is required to be readily available for each SW engineer. Ideally the SW development environment offered by the simulator should be the same as if on the *rhw*.

Some of these requirements are contradictory to each other. For example, cycle-accurate level simulation implies that the simulation speed is slow. Simulating both *simcom*s and application SW at a low level forces the engineers to understand both HW and SW details, which violates the requirement of HW/SW problem isolation. A simulator allowing integration of both cycle-accurate *simcom*s and application SW is typically very expensive and gives an unfamiliar look to SW engineers, which is not able to truly support HW/SW co-development. The conclusion is that a single simulator is not possible to fulfill all these requirements at the same time.

1.5 Summary of Dissertation

The following notations are used in the rest of the document to assist describing the dissertation work.

Notations 1.5.1 *$\mathbb{H}$ is the set of real HW modules to be designed. The number of elements in $\mathbb{H}$ is denoted as $|\mathbb{H}|$ and rhw_i is the i^{th} element in $\mathbb{H}$.*

Notations 1.5.2 *$\forall\ rhw_i \in \mathbb{H}$, $simcom_i$ is a simulation component modeling its behavior and timing characteristics in the specified abstract level.*

Notations 1.5.3 *$\mathbb{SH}$ is the set of simulation HW on which simulation is being carried. The number of elements in $\mathbb{SH}$ is denoted as $|\mathbb{SH}|$ and $simhw_j$ is the j^{th} element in $\mathbb{SH}$. $\forall\ simcom_i$, it is chosen to be simulated on a $simhw$ in $\mathbb{SH}$. If $simcom_i$ is simulated on $simhw_j$, it is denoted as $simcom_i \in simhw_j$.*

Notations 1.5.4 *t_{end} is the time that simulation ends.*

Notations 1.5.5 *$func$ is a function block realizing certain functionality. It will be implemented on a rhw and simulated on the corresponding $simcom$.*

Notations 1.5.6 *T is a task which consists of one or multiple $funcs$ and has its own execution context. Typical examples are SW threads which are managed by the RTOS and HW tasks which are directly managed by the underneath HW. The relationship of $func$, T and $simcom$ is illustrated in Fig. 1.2*

Notations 1.5.7 *$\forall\ func$, an instance of its execution is called a job. $\forall\ simcom_i$, $\mathbb{C}_i$ is the set of job(s) to be simulated on it during $[0, t_{end}]$. The number of job(s) is denoted as $|\mathbb{C}_i|$ and the k^{th} job is denoted as c_i^k.*

Notations 1.5.8 *$\mathbb{R}$ and $\mathbb{N}$ represent the real number set and the non-negative integer set, respectively.*

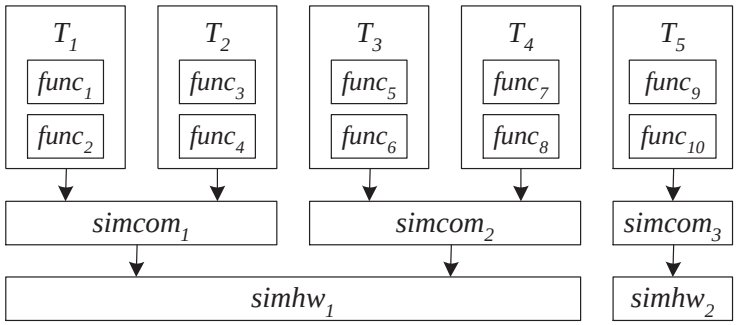

Figure 1.2: $func$, T, and $simcom$

An RTOS modeling tool based on SystemC thread primitive (SC_THREAD) and simulation engine [6] is provided to assist in the generation of an initial specification. Any *simcom* written by C/C++ can also be integrated. The OS model is configurable to support modeling and timed simulation of most existing embedded RTOSes, i.e. Linux [128], DSP/BIOS [15] etc. To model a specific RTOS, a user only needs to: 1) set a few parameters which will be used to configure the RTOS state machine, 2) provide some OS timing information such as scheduler execution delay, context switch delay etc. so that delay annotation can be automatically generated into the model, and 3) "plug in" necessary peripheral driver module(s). The OS timing information can be measured by experiment [167], or estimated by SW estimation techniques [35] [101] [117] [120] if the OS source code is available. Application program timing can adopt a similar strategy and can be isolated from OS timing so that changes to either one will not affect the other unless the HW architecture is changed. The optimized conservative approach is taken to reduce the synchronization overhead. A prediction algorithm is presented to estimate the timestamp of the next OS-wide event for any RTOS with a static or dynamic priority driven scheduler. Compared to [58], the assumption of the prediction algorithm is more realistic. Compared to [79], our tool achieves higher simulation accuracy. Compared to [131] [29] [59] [47], the advantages of the proposed tool are many folds.

For example, OS timing information is easy to obtain; simulation speed is faster; capability is more comprehensive in terms of handling *simcom* synchronization and modeling I/O behavior. Although [59] achieves higher accuracy (5% vs. 15%), our simulation speed is more than 10 times faster.

Two simulators are provided for specification refinement. The first one is a system dataflow simulator (SDFS) and primarily used by the HW engineers. It models the application by a parameter-driven conditional dataflow graph (CDFG) in which each node is a *func* being modeled at transaction level. The HW to be designed ($\mathbb{H}$) is modeled by a configurable HW graph in which each node is a *simcom* modeling the corresponding *rhw* at cycle-accurate level. SDFS takes the application CDFG and HW graph as the input and carries the simulation to catch the detailed HW activities, i.e. bus arbitration. HW engineers are only required to understand the application's CDFG. The parameter-driven feature gives the performance estimation flexibility, i.e. checking the impact of the SDRAM frequency is only a matter of changing one parameter. Modeling the application by the CDFG does not necessarily mean that the application can only be modeled at high level with limited simulation precision. SDFS is designed to allow hierarchical modeling so that any *func* in the CDFG be expanded into several finer-scale *func*s, and thus more details can be incorporated to improve the simulation accuracy. SDFS is readily available because it is implemented in the SystemC language and can be executed on most PCs and workstations. Compared to the trace-driven simulators such as [158] [53] [143], SDFS provides more convincing results with the capability of simulating resource competition scenarios and predicting performance using parameters obtained from legacy application and HW. Compared to [163] which only models application by a DFG only, SDFS achieves higher accuracy by incorporating the application CFG. Compare to [151] which proposed a simple parametric model of execution cost, SDFS has the capability to model pipelined architecture and instruction level parallelism. Compared to [51] [116] which proposed static abstract

models to obtain a lower/upper bound on the system performance, SDFS provides simulation results covering typical resource usage by the application which is more appropriate information driving the architecture design for multimedia systems.

The second one is a real-time simulation platform (RTSP) provided to SW engineers to truly enable HW/SW co-development and co-simulation. To the best of the author's knowledge, it is the only simulator offering the same SW development environment as if the *rhw* was available, while achieving accurate simulation result and fast simulation speed. It is implemented on legacy DSPs as the *simhw* and can be considered as zero cost since many legacy evaluation version module (EVM) boards can be reused for simulation purpose. To simulate the behavior of a *rhw* to be designed, the corresponding *simcom* is constructed running on a legacy DSP with an appropriate share rate. Each legacy DSP has a novel two-level scheduler which makes each *simcom* progress properly so that the simulation is carried at a proportional speed $(1/\gamma)$ of the *rhw* speed. For any job which would finish at time t on the *rhw*, it will finish no later than $\gamma \cdot t + \triangle$ during simulation where $\triangle$ is a bound. Such a feature eliminates expensive synchronization between *simcom*s while still keeps simulation fidelity. RTSP is implemented to allow "plug in" an RTOS model for any *simcom* if necessary. Compared to [113] [172] which proposed the virtual synchronization technique assuming that the output of the simulation SW only depends on event order but not on the arrival time of each event, RTSP does not need such a strong assumption since the scheduler is designed to force each *simcom* to progress properly. [137] also tries to create a SW-development-friendly simulation platform. It requires two host computers. The application SW is simulated on the first one and all the *simcom*s are constructed in SystemC and VHDL and simulated on the other one. The communication between the two hosts is socket-based so that simulating a register-read can take 3.7 ms. The synchronization problem between HW and SW is not addressed. Compared to [137], RTSP achieves higher simulation accuracy by deploying the legacy DSP as the *simhw* and handling synchronization

between *simcom*s properly. RTSP also achieves much higher simulation speed by utilizing SRIO [12] as the much more efficient communication link between *simhw*s.

RTSP and SDFS can be used to complement each other. RTSP carries the simulation at higher level than SDFS and usually cannot capture activities on buses at every cycle. The information collected from SDFS determines appropriate rate settings to affected *simcom*s to compensate the bus activities. RTSP allows SW engineers optimize the algorithm and suggest improvement to HW architecture changes which are fed to SDFS to refine the specification.

Chapter 2

RTOS Modeling

2.1 Introduction

With the appearing of programmable high performance DSPs, more and more system functionalities that used to be realizable by HW are implemented by SW because of its programmable flexibility and lower development cost. On the other hand, with the increasing complexity of embedded devices, it is common to see in the modern embedded system several SW programs running concurrently on a processor managed by an RTOS. Embedded SW typically has real-time constraints to satisfy in addition to functionality requirements. It is important to be able to validate all these properties together as early as possible in the design cycle, and in the context of running the embedded SW on top of an RTOS. Traditionally, designers perform DSE by manually porting the application SW to the target architecture which consumes a lot of time and is often error-prone [176]. An alternative is RTOS modeling. RTOS modeling is a technique that tries to model existing RTOSes within a single environment and captures the abstracted RTOS behaviors at the system level. Ideally, RTOS modeling should fulfill the following requirements:

1. The simulation of the model should be fast enough. It also implies that it

should model the target RTOS at the transaction level. Cycle-accurate or gate level simulation is not only too slow for simulating a meaningful amount of work, but may also not available at the early design stage.

2. The simulation result of the model should be reasonably accurate for DSE.

3. The model can be easily integrated with other *simcom*s under the same simulation framework.

4. The model is configurable and easy to use. Different candidate RTOSes can be evaluated with little or no change to the application programs.

An RTOS modeling tool is presented to assist in generation of the initial specification [96]. It is implemented based on SystemC [6] by employing its thread primitives (SC_THREAD) and simulation engine. The model is configurable to support modeling and timed simulation of most existing embedded RTOSes, i.e. Linux [128], DSP/BIOS [15] etc. To model a specific RTOS, a user only needs to: 1) set a few parameters which will be used to configure the RTOS state machine, 2) provide some OS timing information such as scheduler execution delay and context switch delay etc. so that delay annotation can be automatically generated into the model, and 3) "plug in" necessary peripheral driver module(s).

We believe that timed OS simulation by delay annotation is an appropriate solution for DSE at the early design stage compared to the traditional approach of using an ISS since the former is much faster and able to achieve reasonable timing fidelity. In [150], it was reported that most ISSes simulate one target instruction by every 50-100 host instructions. To make it worse, most ISSes do not have RTOS support. Another drawback of ISS is that it is not able to provide accurate timing estimation to application programs at the early design stage since in most cases application programs have not been implemented or optimized at that time.

The OS timing information can be measured by experiment [167], or estimated by SW estimation techniques [35] [101] [117] [120] if the OS source code is available. In the presented model, a simple yet effective approach is demonstrated to derive the information from benchmark data provided by the OS vendor. The benchmark results are readily available and quite accurate. Application program timing can adopt a similar strategy and can be isolated from OS timing so that changes to either one will not affect the other unless the HW architecture is changed.

One of the problems associated with delay annotation is that the simulation clock advances in macro steps. As a result, the OS being modeled may advance past a timestamp at which it is supposed to handle an input event (interrupt). Bounding the size of the macro clock step to a fixed small value and checking input event more frequently [176] does not completely solve the problem. A more adaptive solution is needed. The need for adaptiveness in choosing step size is particularly important for RTOS simulation since one of its important goals is to identify the interrupt response latency. Without solving the step size problem, it is not possible to decide whether a delay is truly caused by the modeled OS or because of a large clock step advance. To the best of our knowledge, this problem has not been addressed comprehensively so far. We solve the problem by splitting the clock step δ if the current simulation clock $t + \delta > t_{in}$, which is the timestamp when the next input event will happen; the task that advances the clock will only be allowed to proceed to t_{in} and then blocks. By the time the input event is handled and the task is resumed, the remaining amount $(t + \delta) - t_{in}$ will be added to the simulation clock in a similar manner. Details are given in section 2.4.

HW/SW co-simulation often suffers from high synchronization overhead, which gets more severe for co-simulation of interrupt-based systems where interrupt is used as the communication protocol between *simcom*s [174]. In our work, we adopt the optimized conservative approach where each sender *simcom* informs the receiver beforehand when it will send an event. The receiver cannot advance its

clock over this timestamp. A lot of research has been done to estimate the timestamp of the next output event [101] [173]. However, most of them are based on the unrealistic assumption that a SW task executes on dedicated HW which is not applicable to the general case of multiple tasks running concurrently on a processor. [172] presented some results on priority-driven multi-tasking system. However, their work is subject to two assumptions: 1) the task priority is static; and 2) the scheduler is called only when the timer interrupt occurs. In general, an RTOS performs re-scheduling whenever necessary and not only at timer interrupts. In our work, an algorithm is derived to estimate the timestamp of the next OS-wide output event for any RTOS with a static or dynamic priority driven scheduler.

The rest of the chapter is organized as follows. Section 2.2 gives a survey to the related work. The overall OS modeling framework is described in section 2.3. Section 2.4 explains how to derive the timing information and delay annotation interface. Section 2.5 describes the synchronization protocol. Some experimental results are shown in section 2.6. Section 2.7 summarizes the chapter.

2.2 Related Work

Research on RTOS modeling and simulation can be categorized into the following three directions:

- *System Call Translation*, where any system calls from the application being simulated are re-directed into the host OS and executed. This approach is often used by those ISSes without OS simulation ability to verify the functionality of application SW [150] and provides limited timing information. Clearly it is not able to assist in RTOS selection.

- *Native OS*, which compiles the target OS code and executes it on the *simhw*. For example, WindRiver Systems provides VxSim as a simulation model for

its RTOS [25]. It is also able to verify the functionality of application SW. One of its drawbacks is that it does not support modeling of the HW on which the target RTOS runs, and thus the timing of target RTOS is not accurate. Another drawback is that modeling other RTOSes becomes impossible.

- *Virtual OS*, which simulates the functionality and timing of a real OS. Ideally the virtual OS should be configurable to model any existing RTOSes. OS timing can either be achieved by delay annotations inserted into the virtual OS source code, or calculated by an aggregate timing model. For instance, the scheduling delay can be calculated as a function of the number of ready tasks. Commercial tools such as SoCOS [66] and CarbonKernel [3] are available. The main problem of these tools is that it is not easy to use. To model a specific OS, a user often needs to manually "personalize" the virtual OS by inserting appropriate delay annotations. Gerstlauer in [79] presented an RTOS model based on SpecC. The primitives in SepC are used so that the implementation is simple. The application code can be synthesized and compiled to run on the target RTOS by replacing system calls to the OS model with those to the target RTOS. However, the timing accuracy is not satisfactory. [131] presented an RTOS model on top of SystemC. Its focus is only modeling the scheduling algorithm and the process communication mechanism. [29] [59] and [47] proposed the approach combining a SytemC based RTOS model with a bus functional model for HW. Timing is achieved by inserting delay annotations into the SW code; however, none of them shows how to easily get the delay information. The synchronization prediction ability is not available and thus all of them can simulate a processor-only system. [29] [59] did not say how to model the I/O behavior. [47] only models the blocking I/O for device drivers. For [29] [59], both simulation speeds are about 10x slower than the rhw speed with less than 10% error. [47] is 3 orders of magnitude faster than an ISS with

less than 14% simulation error.

Yoo in [176] proposed an idea of automatic generation of OS model. The designer can choose a set of OS services from a library. The service code will be compiled and linked with a micro kernel to generate the RTOS that is executable on the target processor. Delay annotations are automatically inserted in the target OS source code. The main problem associated with this approach is that it can only generate an RTOS from the library written by the author, but not able to model and simulate any other commercial RTOSes.

Our approach is similar to the virtual OS concept. Compared to [66] [3], out tool is easier to use since delay annotations can be inserted to appropriate places automatically. Compared to [131], our tool requires little changes to the application simulation code other than the delay annotation insertion. To model a specific RTOS, the user only needs to set a few parameters for OS state machine generation, and provide OS timing information which is often readily available from the OS and IC vendors. In [131] the application code has to explicitly wait for messages from the OS model which makes the simulation code significantly different from the real code. Compared to [79], our model is also easy to implement by using the SystemC simulation engine and its thread primitives. The timing information derived from benchmark results is accurate enough to assist in the selection of an RTOS for generation of an initial specification. Compared to [131] [29] [59] [47], the advantages of the proposed tool are many folds. For example, OS timing information is easy to obtain; simulation speed is faster; capability is more comprehensive in terms of handling *simcom* synchronization and modeling I/O behavior. Although [59] achieves higher accuracy (5% vs. 15%), our simulation speed is more than 10 times faster.

2.3 RTOS Modeling

2.3.1 RTOS State Machine

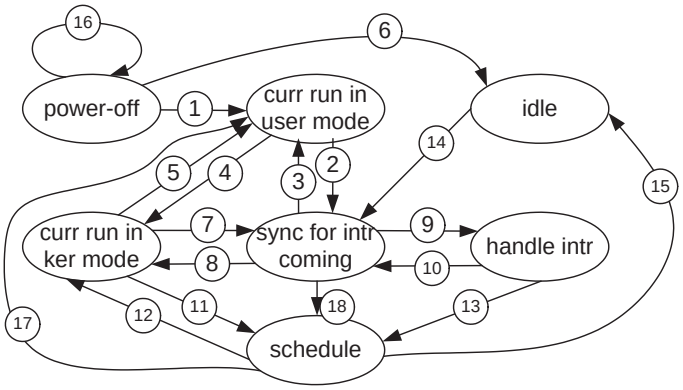

Figure 2.1: Basic RTOS State Machine

A generic RTOS state machine shown in Fig. 2.1 is implemented on top of SystemC. The state machine can be configured to model different RTOSes.

There are seven states in the RTOS state machine, as explained below.

1. *power-off*: the processor has not started running. Before powering on an RTOS, the user needs to connect necessary peripherals such as the timer, to the OS, and also specify a set of parameters to configure the specific state machine for the RTOS being modeled.

2. *curr_run_in_user_mode*: the processor is executing a task in the user mode. In case the professor doesn't differentiate the kernel and user space, the state simply means that a task is executing application code before making a system call into the kernel.

3. *curr_run_in_kernel_mode*: the processor is executing a task in the kernel mode.

4. *sync_for_in_intr*: the RTOS blocks to wait for an external event (interrupt). This is a state that does not exist when the RTOS executes on the *rhw*, but only for simulation synchronization purpose. For optimized conservative simulation, a *simcom* needs to enter this state when it reaches a clock value when an external event may come.

5. *handle_intr*: the RTOS is handling an input event.

6. *schedule*: the RTOS scheduler is selecting a ready task.

7. *idle*: there is no ready task in the RTOS.

Transitions (3) and (8) represent the case where the OS was waiting to receive an event (interrupt) but it did not actually arrive. Note that in practice it is hard to predict the exact timestamp of the next event because of the dynamic behavior of SW. In our model, an event sender tries to predict and notify the receiver the earliest possible timestamp when it will issue an event. When the receiver reaches that time, it waits for the event. If it is sure that the event will not occur because the previous prediction is too conservative, it exits from the synchronization state and continues. The details on the synchronization protocol will be explained in section 2.5.

To model a specific RTOS, a set of parameters need to be set to configure the state machine. The main parameters are: 1) scheduling policy, 2) kernel preemption points (when to make a re-scheduling decision), 3) whether or not to support thread, 4) whether or not to support interrupt thread, 5) IPC mechanism, and 6) whether or not to have memory protection etc.

For example, the embedded Linux implements threads in the kernel and uses threads instead of processes as scheduling entities. It distinguishes three classes of threads: real-time FIFO threads which have highest priority and are not preemptable, real-time round robin threads which have the same priority as real-time

FIFO threads but are preemptable, and timesharing threads which have the lowest priority and are scheduled by a priority aging policy [165]. Interrupt thread is not supported. Interrupt is handled by jumping to the interrupt service routine (ISR). A re-scheduling decision is not made after returning from an ISR unless it is a timer interrupt. Other cases where re-scheduling happens are: current thread blocks; and current thread forks a child thread which will block the parent thread and then start. Linux also supports a rich set of IPC mechanisms such as signals, System V IPC, Unix domain sockets etc.

Take TI DSP/BIOS as another example [15]. It is a single address space OS without virtual memory protection. Processes and threads are not differentiated. It has three classes of threads: HW interrupt threads (ISTs) which have the highest priority and are not preemptable, SW ISTs which have lower priority than HW ISTs and are preemptable, and regular threads which have the lowest priority and are also preemptable. The difference between SW ISTs and regular threads is that SW ISTs run on a common stack while each regular thread has its own stack. The scheduling policy is priority driven without aging. Since IST is supported, an ISR only does a minimum amount of work, and then resumes the corresponding HW/SW IST to make it do the rest work. After returning from an ISR, a re-scheduling decision is always made so that an IST can be started immediately. A re-scheduling is also made when any thread with a higher priority than the current one is resumed. For instance, if the current thread releases a lock, which resumes a thread with a higher priority, a re-scheduling decision is made. The IPCs supported by DSP/BIOS are memory sharing, pipe and mailbox.

2.3.2 OS Sub-Module Design

We adopted the object oriented approach to design the following components in the OS modeling tool: 1) task management, 2) OS kernel, 3) IO module, 4) IPC, 5) timing, and 6) event handling and synchronization protocol. 5) and 6) will be

explained in section 2.5.

Task Management

The task management interface consists of standard routines for

- task creation, i.e. *task_create()*,
- task termination, i.e. *task_exit()* and *task_kill()*,
- task suspension, i.e. *task_yield()* and *task_sleep()*,
- task activation, i.e. *task_resume()*.
- task priority change, i.e. *task_set_prio()*.

Each task is implemented as a SystemC thread (SC_THREAD) with a sc_event object so that context switching can be easily implemented by calling its sc_event methods: *wait()* and *notify()*. To work around the limitation that the SystemC simulation engine does not support dynamic thread creation, a static thread pool is created before the OS is powered on. These threads block on their own sc_event at the beginning. When a *task_create()* call is made, a thread object is retrieved from the pool and a function pointer to the actual thread function is saved in the task object. By the time the thread is scheduled to execute, it unblocks from its sc_event and calls the thread function to start.

If the OS being modeled has memory protection mechanism, only threads belonging to the same process can share the resource. Otherwise, threads and processes are not differentiated.

OS Kernel

The kernel mainly consists of the task scheduler and synchronization objects. We have implemented a flexible priority driven scheduler which supports a mix of preemptive/nonpreemptive scheduling policies with/without priority aging. When a

task is created, three fields in the thread object are specified: *priority*, *aging*, and *preemptable*. The scheduler schedules the tasks based on these fields. For example, to create a DSP/BIOS HW IST, a user can set $priority = 0$ which is the highest priority, $aging = no$ and $preemptable = no$. Associated with each priority is a task queue. Scheduling of the tasks with the same priority can either be FIFO or Round Robin, depending on the configuration specified by the user.

Right now the only synchronization object supported by our tool is the semaphore. Other synchronization objects such as mutex and lock can be derived from semaphore. A semaphore has the following interface to user:

- *sem_init(int count)*, which creates a semaphore with the specified count.
- *sem_inc()*, which increases the count. A blocking thread may be resumed if $count > 0$. Depending on the configuration specified, the order of the threads being resumed can be FIFO or priority based.
- *sem_dec()*, which decreases the count. The calling thread may be blocked if $count \leq 0$.
- *sem_kill()*, which destroys the semaphore object.

Another interface in kernel module is the *power_on()* method which initializes and starts the OS.

IO Module

Modeling IO is a difficult problem because different devices can work in various ways, and each OS has its own IO structure and handles IO in its own way. Modeling individual IO device is not our concern. Instead, our focus is on creating a flexible IO framework so that (1) device models and drivers can be easily plugged in without major modifications, and (2) different IO structures and IO handling mechanisms from different OSes can be easily modeled and simulated in our framework.

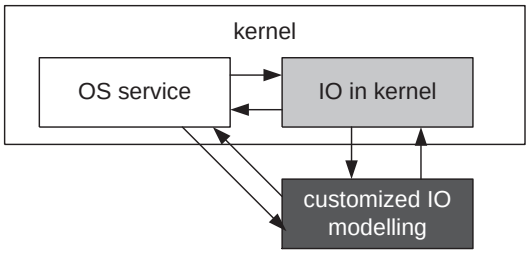

Figure 2.2: IO Module Layer

The overall IO module has two layers as shown in Fig. 2.2. The black block represents the models and drivers of each individual device. The grey block residing in the kernel is the IO interface between applications and device drivers. Driver can request OS service via the IO layer in kernel, i.e. blocking the calling thread, or request the service directly. A typical example of the former case is the DSP/BIOS whose IO manager (IOM) resides in the kernel and manages the state of all tasks requesting IO. Linux is an example representing the latter case where drivers manage the state of the calling thread by themselves.

The IO interface to application program are: *io_open*, *io_close*, *io_read*, *io_write*, and *io_ctrl*.

Each device in the system has a *StructDev* struct as shown in Fig. 2.3. It is shared by both layers. *wQueue* and *rQueue* (line 5-6) are queues saving the pending write/read requests. *drvBuf* (line 8) is the buffer allocated by kernel to the device. *drvStruct* (line 10) is a pointer to specific driver information. *kernelCallback* (line 13) is a function callable by the driver to inform the kernel that an IO request has been finished. It is used by DSP/BIOS. Line 14 - 24 are pointers to the corresponding driver functions to which the IO layer in kernel passes all the requests from applications. The interface between the driver and the kernel is clean but flexible. To model a specific device, the designer only needs to implement these functions in the desired abstract level. *HWIThread* and *SWIThread* are pointers to HW and SW IST respectively if they are used.

```
1   struct DevStruct {
2     char name[20];
3     int mode; // device open mode
4     int state; // state of the device
5     Queue *wQueue; // queue of pending write requests
6     Queue *rQueue; // queue of pending read requests
7     Sem *sem; // semaphore of the device
8     char *drvBuf; // driver buffer and length
9     int drvBufLen;
10    void *drvStruct; // customized driver structure
11    OS *os; // pointer to os
12    PE *device; // pointer to device communicate with
13    FuncPtr kernelCallback; // driver to kernel callback
14    int intrId; // interrupt line used
15    Interrupt_Data *intrData; // interrupt data
16    /* pointers to driver functions */
17    int (*init)( struct DevStruct *dev );
18    int (*open)( DevStruct *dev );
19    int (*close)( DevStruct *dev );
20    void (*write) ( DevStruct *dev, IoStruct *ios );
21    void (*read)( DevStruct *dev, IoStruct *ios );
22    void (*io_ctrl)( DevStruct *dev, IoStruct *ios);
23    void (*destroy)( DevStruct *dev );
24    void (*isr)( DevStruct *dev );
25    Task* HWIThread; // HW interrupt thread
26    Task* SWIThread; // SW interrupt thread
27  }
```

Figure 2.3: Device Struct

Different drivers often have different request formats. Simulating the exact request format for each device is not only unnecessary, but also inconvenient for OS modeling. We provide a unified request format *IoStruct* as shown in Fig. 2.4. It is broad enough to fulfill most drivers' requirement. When sending a request to a specific device, the necessary information from IoStruct is retrieve by the driver. When the request is completed, the return information from the driver is placed in IoStruct.

Line 4 - 9 is the buffer information for the request. For an OS with virtual memory system, an IO request is typically associated with both the buffer in user space and kernel space. *cmd* (line 11) is the command of the request. *commID* (line 12) is used to by multiple *simcom*s sharing the same communication link to identify each other. For instance, if two *simcom*s communicate to each other via Ethernet, it can be used to save the IP address.

Our IO module supports three types of IO operation mode: 1) *synchronous blocking*, 2) *synchronous non-blocking*, and 3) *asynchronous*. The first two modes are common and supported by most OSes. The third one is special yet proved to be efficient both by DSP/BIOS and Linux 2.6. In mode 3) when an application thread issues an IO request, it also provides an application callback pointer (line 13 in Fig. 2.4). If the request cannot be completed immediately, it will be queued to the driver and the application thread can continue doing other work. By the time the request is completed, the driver thread, i.e. HW IST, calls back to kernel which in turn calls the application callback to notify that the request is completed. Inside the application callback, a new request can be issued.

IPC

A rich set of IPCs exist in today's OSes. Modeling each IPC type in its exact form is not only tedious, but also unnecessary. Instead, we abstract them into the following three classes which are able to cover most IPCs.

```
1   struct DevStruct {
2     Task *task; // pointer to calling thread
3     DevStruct *dev; // pointer to the device
4     char *uBuf; // user buffer and length
5     int uBufLen;
6     int uDataLen; // data length in user buf
7     char* kBuf; // kernel buf and length
8     int kBufLen;
9     int kDataLen; // data length in kernel buf
10    int ioMode;
11    int cmd; // request command
12    ID commID; // communication id
13    FuncPtr appCallback; // kernel to app. callback
14  }
```

Figure 2.4: IoStruct

- *Memory sharing* which is the most widely used IPC form in embedded system since most embedded systems have limited memory and a single address space. Memory sharing is easy to implement and also saves the expensive memory resource.

- *Message copying* which is also a commonly used IPC. A message queue protected by a semaphore is created.

- *Signal* which is used by Linux based embedded systems to quickly inform a process to take the desired action. The default actions supported by us are PROCESS_KILL, and PROCESS_BLOCK.

2.4 Simulation Timing

Interface *inc_clock(δ)* is provided to allow the OS clock advance δ units in discrete steps. Fig. 2.5 shows how the simulation clock is incremented without going beyond the timestamp when the next interrupt will occur. If δ is too large, the clock is only advanced to *intrArrTime* (line 4). The OS will handle the interrupt if it actually

```
1  os.inc_clock( δ ) {
2    while( clock+δ > intrArrTime ) {
3      δ := δ - (intrArrTime-clock);
4      clock := intrArrTime;
5      if( intr_arrive() )
6        handle_intr(); // thread may be preempted
7    }
8    clock = clock + δ;
9  }
```

Figure 2.5: Clock Advance

shows up. A re-scheduling decision may be made after handling the interrupt, which may preempt the thread calling *inc_clock()*. When the thread is resumed later, the remaining amount of δ will be accumulated to the OS clock similarly.

Delay annotations are inserted in application and OS code to call *inc_clock()*. Our approach has two advantages compared to the work in [176] [79]. Firstly, simulation can accurately measure interrupt response latency caused by the OS since interrupts are handled at the exact time when they are supposed to be. Secondly, the number of delay annotations can be reduced to alleviate user's work. In [176] [79] since the timing accuracy is directly affected by the clock increase steps and delay annotation frequency, user often needs to manually insert delay annotations in the source code as much as possible to achieve the desired accuracy.

The timing information can be obtained in various ways. A common strategy is to perform compiler analysis on the source code. [101] [117] compile the source code to java byte code, and an estimation is made from the java byte code based on the specific features of the target processor. [35] [120] directly compile the source program to the target instructions to do the timing analysis. Such techniques are applicable to application timing estimation but not to OS modeling since the OS source code is often not available. Wang in [167] proposed an experimental method to measure the OS timing information, i.e. context switch overhead, timer jitter, scheduling cost etc. Such an approach is not applicable unless both the OS binary

and the target HW are available. We propose an idea to derive the OS timing information from benchmark results. The calculation is simple, and the benchmarks are readily available from OS vendors and/or research publications. The timing accuracy is determined by the benchmark data. Since most OS kernels are expected to execute in on-chip memory of embedded processors and directly interface the HW, compared to application programs, the benchmark can be trusted.

Take DSP/BIOS as an example; TI has measured the following timing benchmark on its DSP processors [16]:

- *HW interrupt*, which includes the interrupt latency, interrupt enable/disable costs, interrupt prolog/epilog costs etc.
- *SW interrupt*, which includes SW interrupt enable/disable costs, and the cost of resuming a SW IST.
- *Task*, which includes all kinds of task management costs, i.e. creating a task with/with context switching.
- *Sem* and *Lock*, which includes the costs of operations on semaphores and locks.
- *Memory*, which includes the costs of dynamic memory allocation and deallocation.
- *Pipe* and *Mailbox*, which are the costs of IPC operations.
- *Queue*, which are the costs of all kinds of queue operations.

Other information can be derived from the benchmark results. For instance, suppose it has been measured on the TMS320C64x DSP (on which the I-cache is disabled and the kernel executes in on-chip memory) that creating a task with and without context switch takes 840 and 744 cycles, respectively, then the context switch cost can be calculated as 96 cycles. For another instance, it has also

been measured that it takes 752 cycles from the time when an interrupt occurs to the time after OS handles the interrupt and resumes a new task to run. The 752 cycles consists of interrupt latency, interrupt prolog and epilog, ISR execution, scheduler execution, and task context switch. If the ISR does nothing but increases a semaphore count and the cost of all the other parts are known except for the scheduler execution, the scheduler execution cost can also be calculated.

2.5 Simulation Synchronization

2.5.1 Event Timestamp Prediction

HW/SW co-simulation of interrupt-based systems often suffers from high synchronization overhead. In the worst case when the next interrupt will occur is unknown, simulation has to be carried out in a lock-step manner. Optimized conservative simulation tries to solve this problem by predicting the timestamp of the next event. During the simulation, each *simcom* predicts the timestamp t when it will send out an event and informs the receiver. The receiver makes sure that it will not run beyond t before waiting for the event. A rich set of research results on predicting the event timestamps have been published [101] [173]. Owing to the dynamic behavior of the SW, event time prediction of SW programs is often specified by an interval $[t_{min}, t_{max}]$. The main problem associated with the previous work is that they are all based on the unrealistic assumption that a SW task executes on its dedicated HW. Unfortunately, in most of today's embedded systems this is not the case. When multiple tasks execute concurrently on a processor, the existing research results are still applicable but they are not effective for reducing the synchronization overhead. This issue is illustrated by the following example.

Example 2.5.1 *Suppose task T_1 and T_2 execute on $simcom_1$ and $simcom_2$, respectively. If $simcom_1$ and $simcom_2$ were not shared by any other tasks, T_1 would send*

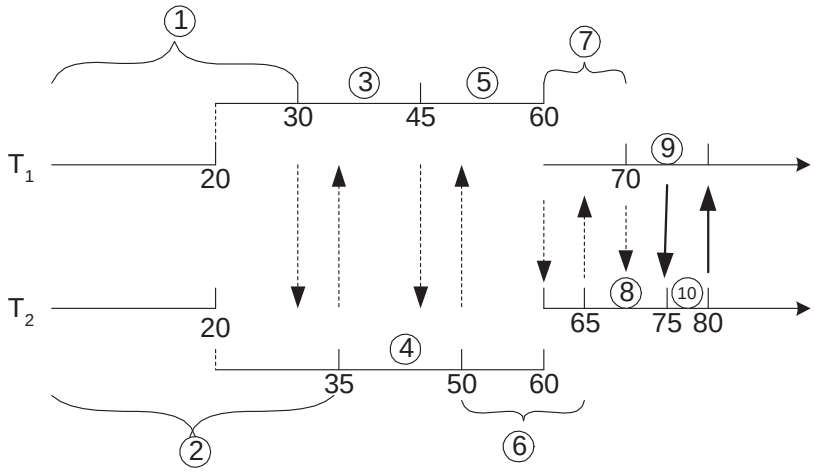

Figure 2.6: Synchronization Overhead

an interrupt to T_2 *at clock* $= 35$*, and* T_2 *would send an interrupt to* T_1 *at clock* $= 40$*. Due to the dynamic behavior of SW programs, we assume that we can only predict that* T_1 *would send the interrupt at either clock* $= 25$ *or* 35*, and* T_2 *would send the interrupt at either clock* $= 30$ *or* 40*.*

Now also assuming both simcoms are shared by other tasks, and both T_1 *and* T_2 *are preempted at* 20 *and resumed at* 60*, the problem is illustrated in Fig. 2.6.*

1. T_1 *is preempted at* 20 *on* $simcom_1$*. The OS on* $simcom_1$ *stops to wait at* 30 *since* T_2 *on* $simcom_2$ *may send an interrupt to it at that time.*

2. T_2 *executes and is also preempted at* 20*. The OS on* $simcom_2$ *has to stop at* 35 *because when* T_1 *will be resumed is unknown, and a safe way is to assume* T_1 *may resume immediately at* 30 *on* $simcom_1$ *which makes it possibly send an interrupt at* 35 *after executing for another* 5 *cycles.*

3. $simcom_1$ *executes other tasks until clock* $= 45$ *due to the same reason as explained in step* 2*.*

4. $simcom_2$ *advances until clock* $= 50$ *due to the same reason as step* 2*.*

```
1   SimTime nextOutEvtTime := MAX_SIMTIME;
2   SimTime preemptInterval := 0;
3   SimTime outEvtTime;
4   for( i = 0; i < MAX_TASK_PRIORITY; i++ ) {
5     Task tsk := prioTaskQueue[i];
6     if( tsk == NULL )
7       continue;
8     outEvtTime := clock + preemptInterval + tsk.nextEvt
9     preemptInterval := preemptInterval + tsk.nextPause;
10    if( outEvtTime < nextOutEvtTime )
11      nextOutEvtTime := outEvtTime;
12  }
```

Figure 2.7: OS-Wide Event Time Prediction

5. *$simcom_1$ advances to* 60 *due to the same reason as step* 2. *When it reaches time* 60 *and* T_1 *is resumed, it becomes known that the nearest time* T_1 *may send an interrupt is at* 65.

6. *$simcom_2$ advances to* 60 *and resumes* T_2. *Then* T_2 *executes until reaching* 65. *It becomes known that the nearest time* T_2 *may send an interrupt is at* 70.

7. T_1 *runs to* 65 *and it becomes known that* T_1 *is taking a longer path and will send an interrupt at* 75. *$simcom_1$ continues executing* T_1 *until reaching* 70.

8. *Similarly, $simcom_2$ executes* T_2 *until* 75. *It becomes known that* T_2 *will send the interrupt at* 80.

9. T_1 *runs to* 75 *and sends the interrupt. It continues until reaching* 80.

10. T_2 *receives the interrupt at* 75. *Then it advances to* 80 *and sends its interrupt to* T_1.

As we can see, the synchronization operations during interval $[20, 60]$ are due to the fact that how long T_1 and T_2 will be preempted is unknown. If such preemption intervals are also predictable, those synchronization actions are not necessary.

We have designed an algorithm to predict the timestamp of the next OS-wide event for the more complicated but realistic case where multiple tasks execute concurrently on a processor, managed by an RTOS. We assume that the RTOS scheduling policy is priority driven, which is true for most RTOSes. Tasks can be preemptive or non-preemptive. Application programs are required to report two time predictions to the kernel: *nextEvt* which is the nearest time when it will issue a system call to cause an output event, and *nextPause* which is the nearest time when it will issue a system call which may make it block, i.e. a blocking I/O call, a *task_sleep()* call etc. Those predictions are made by analyzing the application program, assuming it uses the processor exclusively. The predicted results are relative values, i.e. if the clock is t when the prediction is made, and $nextEvt = \delta$, then the nearest time when the application sends out an event is $t + \delta$.

We provide an interface *reg_prediction(nextEvt, nextPause)* to allow insert predictions dynamically in application programs. The user needs to identify all the paths leading to system calls which either send out events or block the caller. In case the system call is a blocking I/O call, *nextEvt* will be the same as *nextPause*. With the help of some advanced research compiler that can performs the analysis automatically [2], we believe that the speedup gain in the simulation is worth the amount of additional work.

Owing to the page limitation, we only show the prediction algorithm in Fig. 2.7 for the case where the OS scheduling policy is a preemptive priority scheduler and each priority level has at most one task. Other cases can be derived similarly.

For a task with priority i, *preemptInterval* is the interval when it may be preempted by any higher priority tasks each of which will execute for at least its predicted *nextPause* amount of time (line 9). Thus, the earliest time that it can send out an event is $clock + preemptInterval + tsk.nextEvt$ (line 8). The timestamp of the next OS-wide event is the smallest among them. Suppose it is calculated that the next event will be sent out by a task T with priority i, the prediction needs to

be updated whenever any of the following condition becomes true:

- Any task with a priority higher than i is created, resumed, or killed since the last prediction is made.
- Task T changes its priority
- Any other task raises its priority from lower to higher than i, or vice verse.

2.5.2 Synchronization Protocol

We model communications between the event sender and receiver at the transaction level which is appropriate for generating an initial design specification. The interface of the synchronization protocol is given by:

- *send_event()*. This is called by the task which is about to send an event.
- *is_sync()* and *sync_resume()* which are provided by both communication sides to know whether the other one is in synchronization state and resumes it if necessary.
- *get_event()* and *return_ack()* which are provided by the sender but called by the receiver to retrieve the event and return acknowledge state to the sender.

Fig. 2.8 is an illustration of how the protocol works. When a sender is about to issue an event, it prepares the event data and calls *send_event()*. *send_event()* first checks whether the receiver is waiting for the event by calling *receiver.is_sync()*. If it is true, it resumes the receiver by calling *receiver.sync_resume()*. Then the sender blocks to wait for the receiver to retrieve the event.

At the receiver side when its clock reaches the predicted timestamp when an event may arrive, it first checks whether the sender is waiting for sending the event by calling *sender.is_sync()*. If not, it blocks itself. By the time it is resumed by the

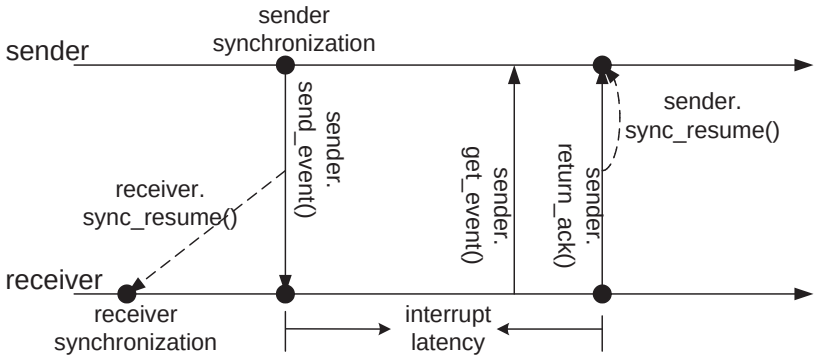

Figure 2.8: Synchronization Protocol

sender, or the sender is already in synchronization state, it calls *sender.get_event()* and *sender.return_ack()* to finish the transaction. The sender resumes itself inside *return_ack()*.

Note that a receiver may enter synchronization state "prematurely" because of the conservative prediction by the sender. In that case when the receiver is resumed by the sender, it needs to advance to the actual clock when the event actually happens before receiving the event.

Deadlocks can happen when multiple *simcom*s wait to receive events from each other, owing to the conservative event time prediction. For instance, if $simcom_i$ predicted to send an event to $simcom_j$ at 10 but actually it will send it at 20, while $simcom_j$ predicted to send an event to $simcom_i$ at 15 but it actually will send it at 30, both $simcom_i$ and $simcom_j$ will wait each other to send an event at 15 and 10, respectively. To avoid deadlock in our tool, whenever a *simcom* is about to wait for an event, it checks whether any other *simcom* with a smaller clock timestamp is waiting for it to send an event. If any of such *simcom* exists, it gives an updated event time prediction and resumes that *simcom*.

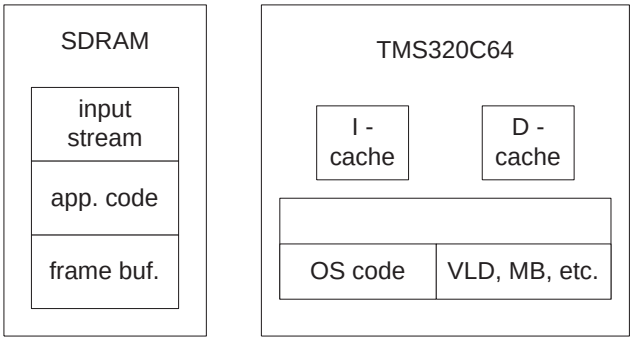

Figure 2.9: H.263 Decoder System

	Worst Frame	Best Frame	Average
Sync. read	39.27 ms	6.64 ms	31.37 ms
Async. read	30.21 ms	4.78 ms	23.07 ms

Table 2.1: Simulation Result of H.263 Decoder

2.6 Experiment

We used an H.263 baseline decoder application from TI [7] to test the effectiveness of our tool. This application runs on a DM642 [21] EVM board which mainly consists of a 600MHz TMS320C64 DSP and 32 MB external SDRAM. The TMS320C64 DSP has a 128-Kbit L1 instruction cache, a 128-Kbit L1 data cache and a 2-Mbit internal memory. The system block diagram is shown in Fig. 2.9. A 4CIF format input stream is stored in the external SDRAM. After a frame is decoded, it is placed in the frame buffer which will be transferred to a monitor to display through a video port. The RTOS is TI DSP/BIOS. The data being placed in internal memory are 1) all VLD tables, 2) zigzag index table, 3) TCOEF length table, 4) reconstructed MB buffer, 5) reference MB buffer, and 6) IDCT buffer. All application and OS code are placed in external SDRAM initially but reside in instruction cache during runtime. All data block transfer and instruction cache filling use DMA. Excluding the H.263 application code, the implementation of the RTOS model and other *simcom* models for related rhw, i.e. DMA consists of 4200+ lines of C++ code.

For each frame we measured the time interval from the point it starts to be decoded to the point when the decoded frame is copied into the frame buffer. The DSP processor transfers the decoded frames to the frame buffer via DMA in asynchronous mode as explained in section 2.3.2. We tested two cases for which the DSP reads input stream either by synchronous blocking mode or asynchronous mode. The results are shown in table 2.1.

As expected, the simulation results show that the asynchronous input read mode is much more efficient than the synchronous blocking mode since IO actions are in parallel with the CPU operations. For the same application, TI has reported that DM642 is able to decode H.263 baseline 4CIF stream at about 20 ms/frame in average. Our simulation results are a little inferior but close to the TI benchmark. The reasons are threefold: 1) inaccuracy of the delay annotation model, 2) inaccuracy caused by the behavior model of the DMA and cache, and 3) data cache is disabled.

The simulation speed of our model is much faster than an ISS. On our workstation with an Intel Cerelon 1.33GHz CPU and 256 MBytes RAM, the simulation of 2×10^9 DSP cycles takes 7.1 seconds, which is about 2.817×10^8 cycles/s. In contrast, TI CCS simulates 7.630×10^4 cycles/s on the same machine. The speedup is more than 3 orders of magnitude in this example.

2.7 Conclusion

An RTOS modeling tool built on top of SystemC [6] is presented to assist in generation of an initial specification. It is configurable to support modeling and timed simulation of most popular embedded RTOSes. Timing is achieved by delay annotations. The OS timing information can be derived from benchmark data provided by the OS vendor or published research results. Since OS code directly interfaces with the HW and typically executes in on-chip memory, the benchmark data can

be trusted with high confidence. Application timing information can be profiled or estimated. It is also isolated from OS timing so that changes to either one will not affect the other unless the HW architecture being modeled is changed.

The optimized conservative approach is taken to reduce the synchronization overhead. The prediction algorithm is based on the more realistic assumption that multiple tasks execute concurrently on a processor, managed by a static or dynamic priority driven scheduler.

One of the problems associated with delay annotations is that the simulation clock advances in macro steps whose granularity may be so large such that the simulation clock may advance beyond the timestamp when an input event is supposed to be handled. Such a problem makes it impossible to measure the interrupt response latency incurred by the OS being modeled. The problem is solved by automatically splitting the clock advance step when the step is too large.

Experiments show that the accuracy of our modeling approach is acceptable. Compared to ISS, the tool can speed up the simulation by more than 3 orders of magnitude.

Chapter 3

System Dataflow Simulator

3.1 Introduction

As industry moves from the general DSP era to the application-specific DSP era, new embedded applications often adopt from the older ones many similar functions but in different combinations and in system architectures that impose more challenging timeliness requirements. In particular, the current design trend is to keep the DSP core unchanged but to re-design the on-chip bus structure and the peripherals, increase the HW frequency and/or add more HW accelerators for the new application. Before the design process starts, it is critical to make sure that the design specification "matches" the application requirements. As being stated in section 1.4.2, a single simulator cannot meet all the requirements of specification refinement. The System Dataflow Simulator (SDFS) being presented is intended primarily to be used by the HW engineers to help generate an accurate HW specification for the targeted application. It models the application by a parameter-driven conditional dataflow graph (CDFG) in which each node is a function block ($func$) being modeled at transaction level. The HW to be designed ($\mathbb{H}$) is modeled by a configurable HW graph in which each node is a $simcom$ modeling the corresponding

rhw at cycle-accurate level. HW engineers are required to understand the application CDFG and the HW graph with more details to carry the simulation. The following are the properties of SDFS.

- *System Level Performance Estimation*

  Assuming that the DSP core and compiler remain unchanged, the functions adopted from the old applications consume the same number of cycle(s) per datum under the same execution condition. SDFS focuses on performance estimation at the system level, taking into account the changes in the execution condition caused by races and different functional compositions.

- *Parameter-Driven*

  The execution cost of each $func$ in the application CDFG is modeled by user-configurable parameters, which allows highly flexible performance estimation. For instance, to see the impact of the burst-read size on system performance, a user only needs to change one parameter and check the simulation result.

- *Hierarchy Modeling*

  One thing to note is that the CDFG approach does not necessarily mean that the application can only be modeled at high level with limited simulation precision. The hierarchical modeling feature allows any $func$ in the CDFG be expanded into several finer-scale $func$s, and thus more details can be incorporated to improve the simulation accuracy.

- *Low Level Simulation*

  The simulation is carried out at the cycle-accurate level in order to capture the detailed activities on the HW, i.e. bus arbitration.

- *Flexible Probing*

SDFS allows the insertion of various probes in the application CDFG and HW graph to collect interested information.

The experimental results show that SDFS is not only able to accurately estimate the system performance, but also help identify the system bottleneck and find the optimal SW implementation solution. Such information helps the designer to optimize the system architecture although the tool does not generate an optimal architecture specification by itself.

The rest of the chapter is organized as follows. Section 3.2 gives a survey to the related work. The details of the tool are described in section 3.3. Section 3.4 gives an example showing how to use the tool. Some experimental results are shown in section 3.5. The conclusion and future work are summarized in section 3.6.

3.2 Related Work

A number of simulators for IP-based systems are found in [158] [53] [143]. Usually a trace file abstracting the application behavior is derived from the application and fed into the simulator. A task is assumed to consist of multiple functions each of which has a known execution cost. The common drawbacks of these simulators are: 1) a fixed application trace does not provide convincing estimation results. For example, video clips with different motion activities consume very different numbers of DSP cycles for encoding/decoding. Moreover, a minor change to the application behavior requires the derivation of a new trace which is time-consuming. 2) The performance prediction precision is limited because the function execution costs are assumed to be fixed even though the execution conditions may change. For these simulators, the execution cost needs to be profiled for each function in the same application running on the same HW platform in order to estimate the performance of an application on the HW platform. Ideally, feeding the execution costs to the simulator should yield very accurate results without surprises, since the function execution conditions

remain the same. However, none of the mentioned work has reported any results that pertain to changes to the HW architecture and/or application behavior, i.e. other than raising the DSP core frequency, everything else remains as same.

[163] proposed a simulator focusing on estimating performance changes caused by different bus structures. The results are not accurate because of the following limitations: 1) it models the application by a dataflow graph only which does not support conditional branches and thus cannot truly capture the application behavior; 2) the activities on HW, i.e. SDRAM accesses, are simulated at the macro block level which only captures the performance of burst accesses but not that of sporadic ones.

[151] proposed a parametric model in which the execution cost of a $func$ is the summation of three portions: *core processor time*, *instruction fetch time*, and an optional *memory access time*. This approach is not able to model the pipeline architecture of modern DSPs in which instruction fetching and memory access are performed in parallel with instruction execution and multiple instructions can be executed concurrently on different core units.

[51] [116] proposed static abstract models that try to obtain a lower/upper bound on the system performance. Such information does not reflect the typical resource usage by the application and thus not appropriate for driving the architecture design of multimedia systems. Besides, not all application behaviors can fit into their models and the static queuing model does not always abstract the actual application's behavior. Another problem is that some resource competition conditions cannot be captured, i.e. when both the DSP core and the DMA are accessing the internal memory at the same time.

SDFS solves the problems mentioned above and thus is able to assist in specification refinement with high confidence.

3.3 Tool Description

3.3.1 Application Dataflow Graph

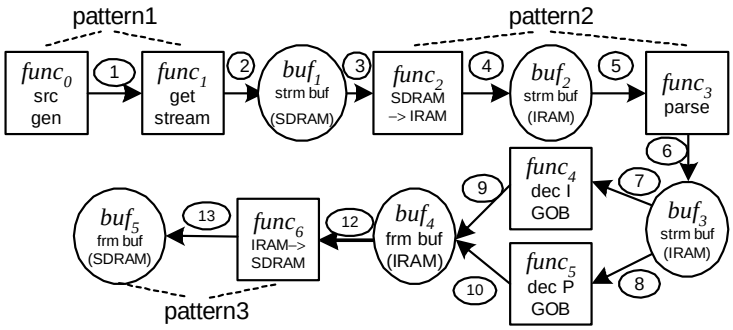

Figure 3.1: Application DFG Example

An application dataflow graph (DFG) is a triple $\langle func,\ DE,\ buf \rangle$. An example is shown in Fig. 3.1.

- Each $func$ is a functional block in the application shown as a rectangle. A $func$ may execute multiple times at run time; each execution instance is called a *job*. A job may consist of multiple phases each of which involves reading and processing input data, and produces the output data at the end. We call each phase a *sub-job*.

- Each DE shown as an arrow is the data edge and denotes the dataflow in the direction of the arrow.

- Each buf shown as a circle denotes the virtual buffer used by $func$s to store data.

The DE connections in an application DFG conform to one of the following three patterns, as shown in Fig. 3.1.

- $func_1 \xrightarrow[DE]{} func_2$ shown as *pattern1*: It means that when $func_1$ transfers a datum to $func_2$ via DE, the transfer is synchronous meaning that $func_1$ blocks until $func_2$ has received it.

- $func_1 \xrightarrow[DE_1]{} buf \xrightarrow[DE_2]{} func_2$ shown as *pattern2*: It means that the data transfers are asynchronous. The data are stored by $func_1$ into buf via DE_1 and later read by $func_2$ via DE_2.

- $func_1 \xrightarrow[DE]{} buf$ shown as *pattern3*: It means that $func_1$ stores the output data to buf via DE, and the data will not be read by any other $func$.

We shall adopt the following naming convention.

- A DE pointing to (originating from) a $func$ is called *input DE* (*output DE*) of the $func$. A $func$ can have multiple input and output DEs.

- A $func$ at the originating (arrow) end of a DE is called *source func* (*sink func*) of the DE.

- A DE pointing to (originating from) a buf is called *input DE* (*output DE*) of the buf. The $func$ originating from (pointed by) the input DE (output DE) is called *source func* (*sink func*) of the buf. A buf has exactly one input DE and at most one output DE.

3.3.2 Application Conditional-Flow Graph

Fig. 3.2-(a) shows an application conditional-flow graph (CFG) in which each node represents a $func$ and each directed edge is a condition edge (CE). The configuration of CEs determines the order in which each $func$ is executed. The $func$ at the originating (arrow) end of a CE is called *source func* (*sink func*). When the condition is evaluated to be true, the sink $func$ is ready for execution.

SDFS supports the following three types of CEs that are sufficient to model most multimedia applications.

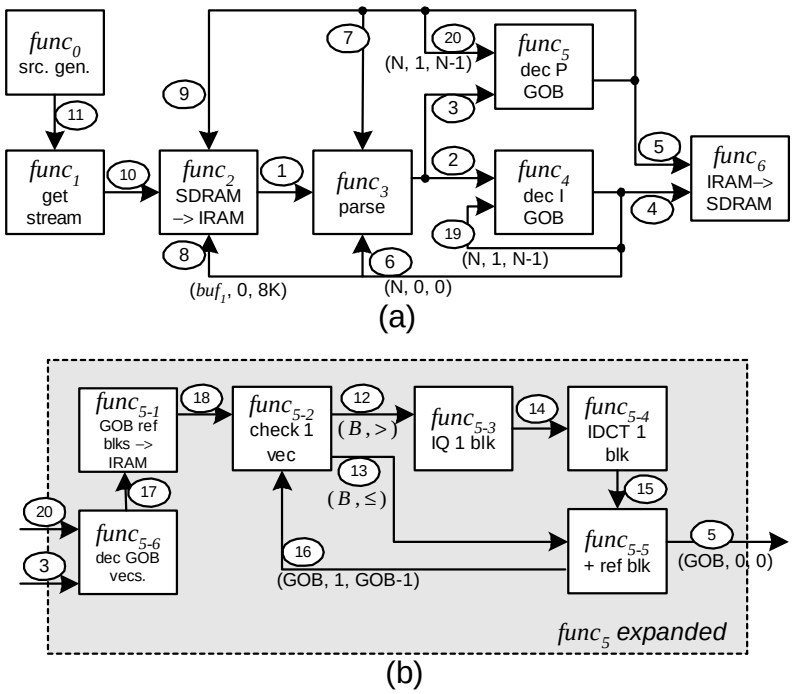

Figure 3.2: Application CFG example

- *cIdxCE* evaluates the condition based on the source $func$'s job index ($cIdx$). Its parameters are: ($period$, $residualStart$, $residualEnd$) which are all integers and

$$0 \leq residualStart \leq residualEnd \leq period \tag{3.1}$$

The condition becomes true when

$$cIdx\%period \in [residualStart, residualEnd] \tag{3.2}$$

Take CE_6 in Fig. 3.2-(a) as an example. Since $period = N$ and $residualStart = residualEnd = 0$, the condition is true only if $cIdx = k \cdot N$ ($k \in \mathbb{N}$), which means that $func_3$ is executed once every N times after $func_4$ is finished. Correspondingly, CE_{19} means that $func_4$ repeats executing itself in other cases.

- *bufDataCE* evaluates the condition based on the amount of data in the corresponding buf. Its parameters are (buf, $dataNumLow$, $dataNumHigh$). The condition is true if

$$n \in (dataNumLow, dataNumHigh) \tag{3.3}$$

where n is the number of datum in buf. For example, for CE_8 in Fig. 3.2-(a), $func_2$ is executed after $func_4$ when the amount of data in buf_1 is less than $8K$ bytes.

- *randCE* evaluates the condition based on the value of a randomly generated real number r ($r \in [0, 1]$) and the parameter pair ($threshold$, $operator$) where $operator$ can be $>$ or $\leq$.

In the case when $operator$ is $>$, the condition is true if

$$r > threshold \tag{3.4}$$

In the case when $operator$ is $\leq$, the condition is true if

$$r \leq threshold \tag{3.5}$$

For example, $func_{5-3}$ and $func_{5-5}$ in Fig. 3.2-(b) are executed respectively based on if $r > B$ or $r \leq B$.

A source $func$ can have multiple CEs originating from it. Each CE is evaluated when a job is finished. A sink $func$ can also have multiple CEs pointing to it and its firing condition is a logical combination of them. To automate the evaluation of firing conditions, such CEs can be grouped into a set of *condition patterns* each of which is true only if all of its CEs are true. A $func$ can be fired if any one of its condition pattern(s) becomes true. For example, $func_6$ in Fig. 3.2-(a) has CE_4 and CE_5 pointing to it. By putting them into 2 separate patterns, $func_6$ can be fired after either $func_4$ or $func_5$ is finished.

3.3.3 HW Graph

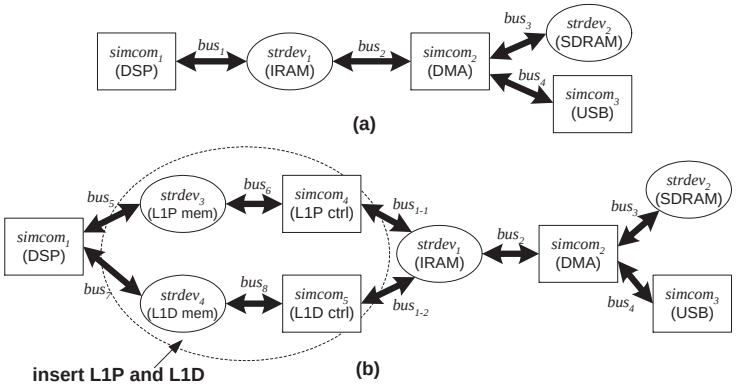

Figure 3.3: HW Graph Example

A HW graph is a triple $\langle simcom, bus, strdev \rangle$. An example is shown in Fig. 3.3.

- A *simcom* shown as a rectangle models a rhw which has the ability to read/write and process data. Typical examples are DSP, DMA etc.
- A *bus* shown as a bidirectional edge is a link along which data can be transported in either direction.
- A *strdev* shown as a circle is the storage device used for storing the application data. Typical examples are SDRAM, internal memory (IRAM) etc.

A *bus* can either connect a *simcom* with a *strdev* or connect two *simcom*s. The former represents the case where the *simcom* can read/write data from/to the *strdev*. The latter represents the case where two *simcom*s can communicate synchronously via the *bus*. A *strdev* can have multiple *bus*es connected to it. Access requests from those *bus*es are served in the first-in-first-out (FIFO) order.

3.3.4 Application CDFG & HW Parameters

The application CDFG and the HW graph represent the application's dataflow behavior and HW architecture respectively, while their parameters specify the system performance requirements. The key parameters are listed below.

$func$ Parameters

- *property* can either be PERIODIC to indicate that a $func$ is executed periodically, or REACTIVE in which case a job is fired if at least one of its conditions pattern is true.
- *initCond* specifies whether the first job of the $func$ can be fired or not at the beginning of the simulation, regardless of its condition pattern.
- *priority*: we assume each *simcom* has a scheduler that schedules its $func$s based on their priorities. For a DSP, it can be a mimic of the RTOS scheduler. For a DMA, it can be the internal HW arbitrator.
- *numSubjob* specifies the number of sub-jobs for each job.
- *inDataPerSubjob* and *outDataPerSubjob* specify a $func$'s I/O behavior. A job consists of *numSubjob* sub-jobs each of which reads *inDataPerSubjob* amount of data from each input DE, processes the data, and writes *outDataPerSubjob* amount of data to each output DE.
- *costPerByte* specifies the average number of cycle(s) required to process each byte for this $func$.
- *pipeline* specifies whether the I/O operation can be executed in parallel with the data processing operation.

DE **Parameters**

- *dataWidth* specifies the unit of each data transfer on the *DE*.
- *readType* defines the read operation property on the DE. It can either be LOOKUP or CONSUME. The latter decrements the amount of data in the *buf* for each read operation while the former does not.
- *writeType* defines the write operation property on the *DE*. It can either be OVERWRITE or PRODUCE. The latter increments the amounts of data in the *buf* for each write operation while the former does not.

simcom **Parameters**

- *frequency* is the clock frequency (in MHz) of the *rhw* being modeled by the *simcom*.
- *engineNum* specifies the maximum number of jobs that can be executed in parallel on the *rhw*. For a DSP core, it is set to 1. For a DMA device, it is set to the number of parallel channels being supported.
- *syncCost* specifies the number of cycles to handle an incoming synchronous datum/signal.

strdev **Parameters**

- *frequency* is the clock frequency (in MHz) of the storage device being modeled.
- *busWidth* is the width of the *bus* connecting with the *strdev*. By changing this parameter, the simulation result reflects the impact of the bus width to system performance.
- *cycPerRead* and *cycPerBurstRead* specify the cost of a single and burst read operation, respectively.

- *cycPerWrite* and *cycPerBurstWrite* specify the cost of a single and burst write operation, respectively.

```
pipeline = false;
for ( int i = 0; i < numSubjob; i++ ) {
  read ( inDataPerSubjob, i ); // read for subjob_i
  process ( inDataPerSubjob, i ); // process for subjob_i
  write( outDataPerSubjob, i ); // write for subjob_i
}
```

(a) Non-Pipeline DSP

```
  pipeline = true;
  read ( inDataPerSubjob, 0 ); // read for subjob_0
  process( inDataPerSubjob, 0 ); // process for subjob_0
|| read ( inDataPerSubjob, 1 ); // and read for subjob_1
for ( int i = 1; i < numSubjob-1; i++ ) {
  write( outDataPerSubjob, i-1 ); // write for subjob_{i-1}
|| read ( inDataPerSubjob, i+1 ); // and read for subjob_{i+1}
|| process ( inDataPerSubjob, i ); // and process for subjob_i
}
  process( inDataPerSubjob, numSubjob-1 );
|| write( outDataPerSubjob, numSubjob-2 );
  write( outDataPerSubjob, numSubjob-1 );
```

(b) Pipeline DSP, (numSubjob > 1)

Figure 3.4: Modeling Pipeline/Nonpipeline DSP

The pseudo code in Fig. 3.4-(a) shows how to model the cost of a $func$ execution on a DSP with non-pipeline architecture. The $func$ has an input DE and an output DE. For each sub-job, the $func$ reads $inDataPerSubjob$ data, process them and writes $outDataPerSubjob$ data. The total cost is the summation of all the read, process and write costs.

The pseudo code in Fig. 3.4-(b) shows how to model the execution cost for the same $func$ on a DSP with pipeline architecture. Inside the "for" loop, the read operation for $subjob_{i+1}$, the processing operation for $subjob_i$ and the write operation

for $subjob_{i-1}$ are parallelized to model the pipelining feature. The cost of each loop iteration can be represented as $\max\{t_r, t_p, t_w\}$, where t_r, t_p and t_w is the cost of read, process and write operation, respectively.

The cost of each processing operation can be calculated as

$$t_p = costPerByte \cdot dataWidth \cdot inDataPerSubjob \tag{3.6}$$

The cost of each read and write operation depends on the *strdev* being accessed. A sequence of read/write operations to the same *strdev* are assumed to be burst accesses. The following equations model t_r and t_w.

$$\begin{aligned} n_r &= \lfloor \tfrac{dataWidth}{busWidth} \rfloor \cdot inDataPerSubjob \\ t_r &= tr_{blk} + cycPerRead + cycPerBurstRead \cdot (n_r - 1) \end{aligned} \tag{3.7}$$

$$\begin{aligned} n_w &= \lfloor \tfrac{dataWidth}{busWidth} \rfloor \cdot outDataPerSubjob \\ t_w &= tw_{blk} + cycPerWrite + cycPerBurstWrite \cdot (n_w - 1) \end{aligned} \tag{3.8}$$

where tr_{blk} (tw_{blk}) represents the read (write) blocking time due to resource competition and its actual value is captured during simulation.

3.3.5 Mapping Application DFG to HW Graph

The following rules define how to map an application DFG to a HW graph.

- $func \xrightarrow[DE]{} buf \Rightarrow simcom \xrightarrow[bus]{} strdev$, in which case $func$, DE and buf are mapped to *simcom*, *bus* and *strdev* respectively.
- $buf \xrightarrow[DE]{} func \Rightarrow strdev \xrightarrow[bus]{} simcom$, in which case buf, DE and $func$ are mapped to *strdev*, *bus* and *simcom* respectively.
- $func_1 \xrightarrow[DE]{} func_2 \Rightarrow simcom_1 \xrightarrow[bus]{} simcom_2$, in which case $func_1$ and $func_2$ are mapped to $simcom_1$ and $simcom_2$ respectively, and they communicate synchronously with each other via the *bus*.

- $func_1 \overrightarrow{_{DE}} func_2 \Rightarrow simcom$, in which case both $func_1$ and $func_2$ are mapped to the same *simcom* and communicate with each other by some SW mechanism so that DE does not need to be mapped to any physical *bus*.

Multiple $func$s can be mapped to the same *simcom*. They are scheduled based on their priorities. Multiple bufs can be mapped to the same *strdev* and they are assumed not to overlap. A *bus* can also have more than one DEs mapped to it. In case multiple DEs compete for the same *bus* at the same time, the arbitration is done as follows:

- If the *bus* connects a *simcom* and a *strdev*, which implies that the multiple $func$s connecting to the competing DE are mapped to the same *simcom* and try to transfer data over the *bus*, then the arbitration is done by the *simcom* based on the priorities of the $func$s.

- If the bus connects two *simcom*s and the competing $func$s are on the same *simcom*, the arbitration is done in the same way as above. This scenario happens when multiple $func$s execute simultaneously on the *simcom*, i.e. several DMA channels are accessing the same *bus*.

- If the *bus* connects two *simcom*s and the competing $func$s are on different *simcom*s, the data/signal transaction is done based on the synchronous communication protocol between the two *simcom*s.

3.3.6 Simulator Implementation

SDFS is implemented on top of SystemC 2.0 [6]. Any windows/Linux PC or workstation can be used as *simhw*. It consists of the following types of modules: $func$, DE, CE, buf, *simcom*, *bus*, *strdev*, *backplane* and *probepoint*.

The main responsibility of a *simcom* is to dispatch the $func$(s) mapped to it and to manage its own clock. When a job is to transfer data, it notifies its *simcom*

which will then interact with the appropriate DE to handle the request. The DE splits the request if the data unit is larger than the *bus* width or the length is larger than the allowed burst size. The request(s) will be submitted to the *bus* to which the DE is mapped and finally reaches the corresponding *strdev*. When the operation is finished, the *simcom* updates its clock. When a data processing operation is finished by a sub-job, it also notifies the *simcom* to update the clock.

The *backplane* synchronizes all the *simcom*s during the simulation so that all the events are processed in a causal order. Although [174] [96] demonstrated that the optimized conservative approach is effective in reducing synchronization overhead for high level simulation, we adopt the conservative approach in our tool due to the following two reasons. 1) For video applications, the amount of data movement is relatively large. Since all data movements need to be simulated at cycle accurate level, the event prediction overhead outweighs the benefit to be gained. 2) The application is modeled at transaction level with several parameters and simulating the behavior of each $func$ requires a small number of CPU instructions. Therefore, the overall simulation speed is still fast enough even though the conservative approach is taken.

HW and SW probepoints are implemented to collect simulation information. By inserting a pair of SW probepoints in the $func$ boundaries of the application CFG, a user can get the best/worst/average execution time between the two points. A HW probepoint can be inserted into a *simcom* to compute its utilization factor. Such type of information helps to identify HW bottleneck(s) and to find the optimal SW implementation solution.

3.4 Example

The following two steps must be completed before a simulation can start: 1) creating and configuring the application CDFG and HW graph; and 2) mapping the

$simcom_1$	$simcom_2$	$simcom_3$	$strdev_1$	$strdev_2$
DSP	DMA	USB	IRAM	SDRAM

(a) Modeling DM642 with Cache Being Ignored

$simcom_4$	$simcom_5$	$strdev_3$	$strdev_4$
L1P cache controller	L1D cache controller	L1P cache memory	L1D cache memory

(b) Modeling DM642 with L1P and L1D being modeled

Table 3.1: Modeling TI DM642

application DFG to the HW graph.

Creating and configuring the HW graph can be done by the HW engineers based on the proposed HW architecture. For example, Fig. 3.3-(a) and Tab. 3.1-(a) show how to model a TI DM642 DSP [21] based system with caches being ignored. Fig. 3.3-(b) and Tab. 3.1-(b) show how to model the L1 program and data cache.

Creating and configuring the application CDFG can be made either by HW or SW engineers based on the desired application behavior. For instance, Fig. 3.2-(a) and Fig. 3.1 show the CFG and the DFG modeling a TI H.263 decoder [7]. The decoder receives the input stream from USB and sends the decoded frame to the frame buffer in SDRAM. Each $func$ and its mapping are described in Tab. 3.2.

func	**mapping**	**description**
$func_0$	USB	Stream source generation
$func_1$	DSP	Receiving stream
$func_2$	DMA	Moving data from SDRAM to IRAM
$func_3$	DSP	Parsing frame head
$func_4$	DSP	Decoding I group of blocks (GOB)
$func_5$	DSP	Decoding P GOB
$func_6$	DMA	Moving decoded GOB to SDRAM
buf_1	SDRAM	Stream buffer storing input data
buf_2	IRAM	Streaming buffer in IRAM
buf_3	IRAM	Streaming buffer without frame head
buf_4	IRAM	Frame buffer storing decoded frame
buf_5	SDRAM	Frame buffer storing decoded frame

Table 3.2: Constructing CFG for H.263 Decoder

func	mapping	description
$func_{5-1}$	DMA	Move GOB reference blocks to IRAM
$func_{5-2}$	DSP	Parse a motion vector
$func_{5-3}$	DSP	I-Quantization a macro block
$func_{5-4}$	DSP	IDCT a macro block
$func_{5-5}$	DSP	Add reference block with IDCT result
$func_{5-6}$	DSP	Decoding GOB motion vectors

Table 3.3: Decoding a P GOB by H.263 Decoder

Obviously the H.263 decoder modeled by the CDFG mentioned above is at a too high a level for obtaining simulation result of sufficient accuracy. Fig. 3.2-(b) and Tab. 3.3 show how $func_5$ which decodes a P group of blocks (GOB) can be expanded to reveal several finer-scale $func$s and achieve more accurate simulation results. Correspondingly, the application DFG needs to be expanded too.

The CEs in the application CFG are configured based on the application behavior. For CIF (352×288) video format, TI H.263 decoder decodes 22 macroblocks (MBs) at a time as a GOB and then moves the decoded GOB to the frame buffer by DMA. There are 18 GOBs in each frame. Therefore, GOB in Fig. 3.2-(b) is set to 22, and N in Fig. 3.2-(a) is set to 18. The parameters in the HW graph are configured based on the proposed HW architecture.

The $func$ parameters can be obtained from the published benchmark [19]. However, it is important to notice that the benchmark results are measured under the best execution condition in the sense that the code reside in L1 program cache and all the data are in L1 data cache. To accurately estimate the system performance using those benchmark data, $func$s modeling the cache activities need to be inserted into the application CDFG at appropriate places. For example, Fig. 3.2-(b) can be further expanded as Fig. 3.5. The behaviors of the inserted $func$s are described in Tab. 3.4. Inserting the cache activity $func$s at appropriate places requires expertise knowledge of SW implementation. Correspondingly, the HW graph of Fig. 3.3-(a) needs to be expanded to Fig. 3.3-(b) to include all the *simcoms* modeling the rhws

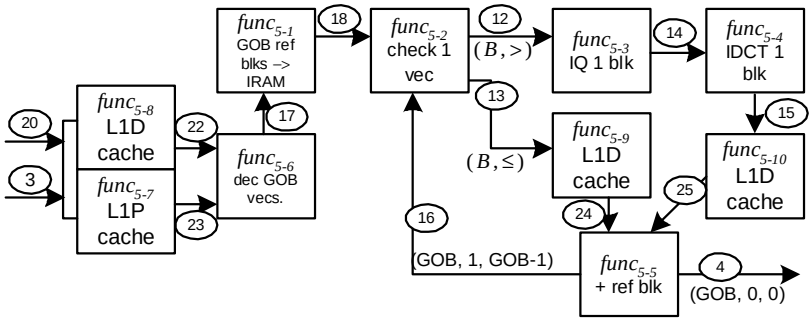

Figure 3.5: Decoding P GOB, Modeling Cache

func	**mapping**	**description**
$func_{5-7}$	L1P	Move code of $func_{5-2,-3,-4,-5,-6}$ from SDRAM to L1P memory
$func_{5-8}$	L1D	Move GOB blocks of coded data from IRAM to L1D memory
$func_{5-9}$	L1D	Move 1 reference block from IRAM to L1D memory
$func_{5-10}$	L1D	Move 1 reference block from IRAM to L1D memory

Table 3.4: Cache Activities for Decoding P GOB

related to cache.

3.5 Experiment

In the first experiment, we tried to simulate the TI H.263 decoder performance on the TI DM642 processor. All the HW parameters are configured based on the DM642 datasheet [21]. All the $func$ parameters are obtained by benchmarking the decoder running on DM642. The purpose of this experiment is not to estimate the performance of a new application on a new rhw but to validate our simulator as

core frequency (DSP+cache)	600MHz
on-chip HW (IRAM+DMA)	300MHz
bus between DSP and cache	256 bits
other buses	64 bits

Table 3.5: TI DM642 Parameters

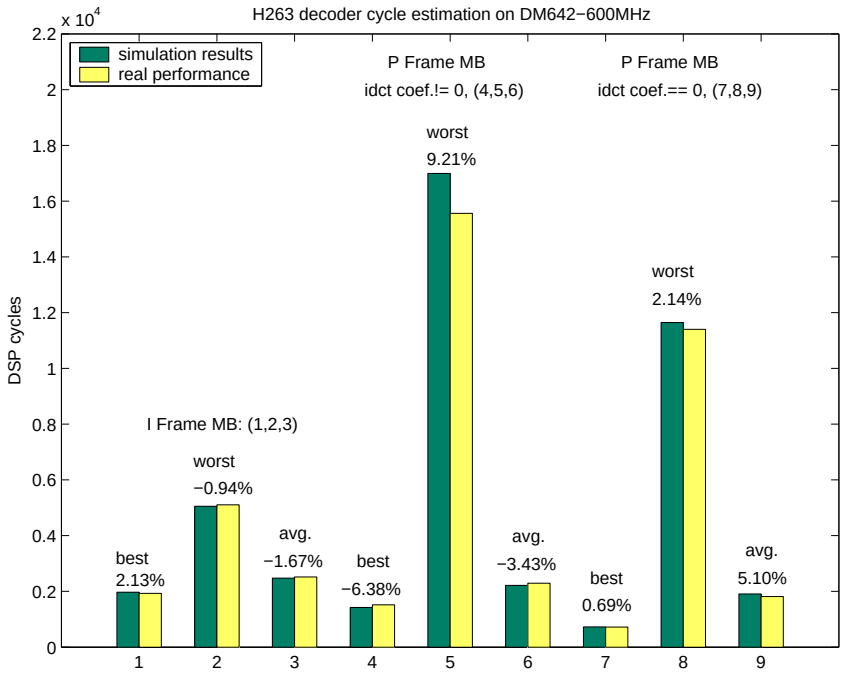

Figure 3.6: H.263 Decoder on DM642-600MHz

was done in [158] [53] [143]. It is expected that the simulation result should be close to the profiling results. The benchmark obtained is the best-case data when both program and data are in L1 cache. $funcs$ modeling cache activities have been inserted into the application CDFG at appropriate places, as in Fig. 3.5. The video format is progressive mode with D1 resolution. The HW architecture is the same as Fig. 3.3-(b) and the main parameters are shown in Tab. 3.5. Since different streams with different scenes and motion activities can require drastically different decoding time per frame, we chose to compare the decoding time per MB which does not quite depend on the input stream. We note that such a comparison actually imposes a much bigger challenge because we have to check whether the estimation results match the real performance on a per MB basis instead of on a per frame basis for which the decoding time of 1350 MBs is summed.

From Fig. 3.6 to Fig. 3.9, the Y-axis is the number of DSP clock cycles,

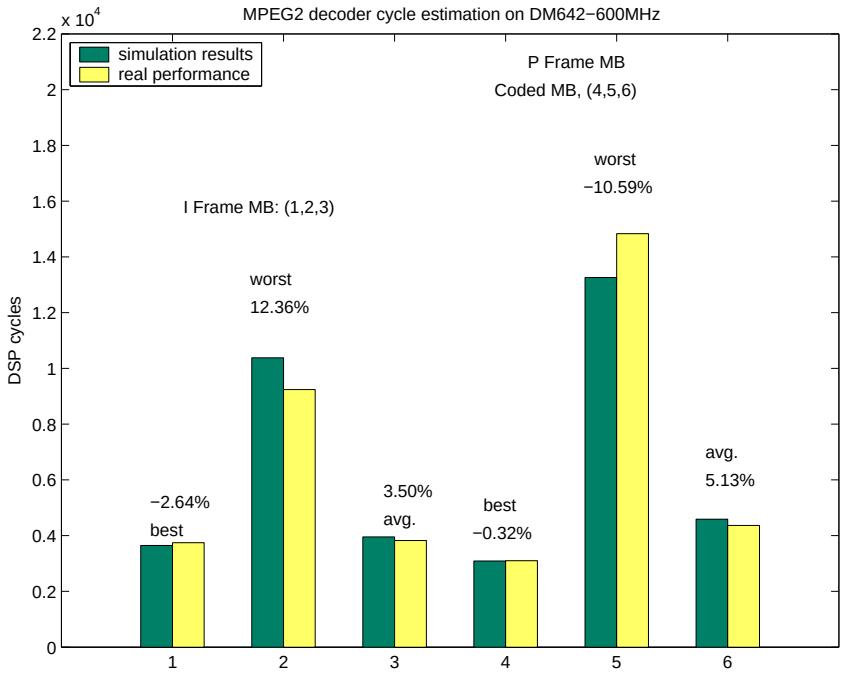

Figure 3.7: MPEG2 Decoder on DM642-600MHz

and the X-axis is the index of the specific case in an experiment. In each grouped bar, the green one represents the simulation result, while the yellow one represents the actual execution result. For example, the case 1 (x=1) in Fig. 3.6 shows the best performance to decode a MB in an I frame for a H.263 decoder running on a 600MHz DM642 chip, and the error between the simulation and actual result is 2.13%.

Fig. 3.6 shows that the simulation results match well with the real performance not only in the best cases, but also in the average and worst cases. Since all the non-best cases are caused by resource competition, i.e. two *simcom*s accessing IRAM at the same time, it is assuring to see that the simulator is able to capture the impact of these race conditions.

In the second experiment, we tried to estimate the performance of a TI MPEG-2 decoder running on the same HW. Compared to H.263 decoder, this one

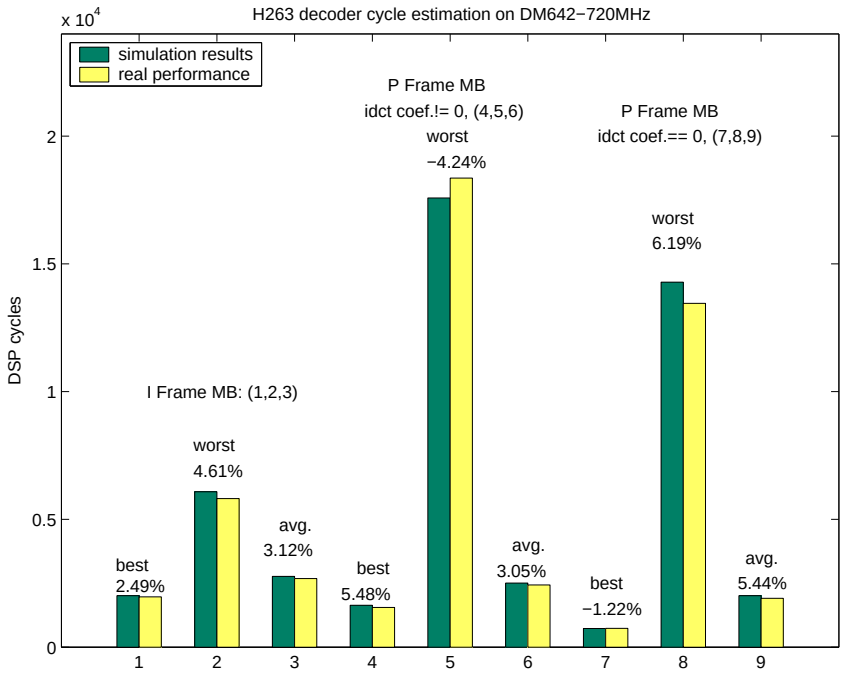

Figure 3.8: H.263 Decoder on DM642-720MHz

has a different CDFG but adopts the same set of key functions when decoding a progressive mode stream. The benchmark data obtained from the first experiment are reused to configure the $func$ parameters. The objective of this experiment is to demonstrate the estimation accuracy for a new application running on the old rhw.

In the third experiment, we raised the DSP frequency and tried to estimate the performance of the H.263 decoder again. The frequency for the DSP core and on-chip HW is raised to 720MHz and 360MHz, respectively. The SDRAM frequency is still 133MHz. The purpose is to demonstrate the estimation accuracy for an old application running on a new rhw.

In the fourth experiment, we tried to estimate the performance of the MPEG-2 decoder running on the DSP at a higher frequency. The purpose of this experiment is to demonstrate the estimation capability of the tool for a new application running on a new rhw.

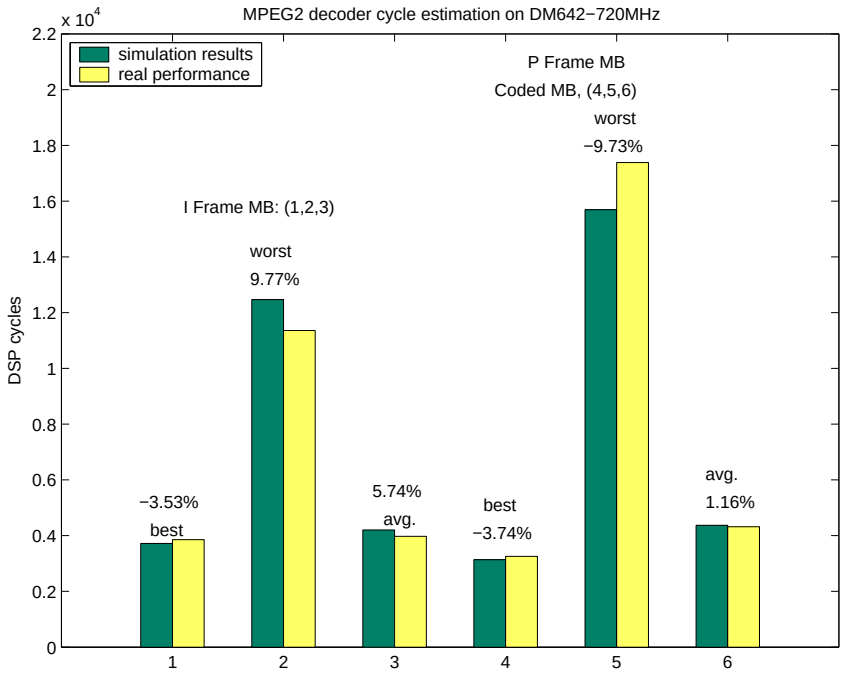

Figure 3.9: MPEG2 Decoder on DM642-720MHz

Fig. 3.7, 3.8 and 3.9 show that the simulation results still match well with the real performance for the best and average cases when the application CDFG and/or HW are changed. The worst case estimations have gone up to 13% error which is mainly caused by the inaccuracy of the cache model.

The simulator is also able to suggest better SW implementation solutions. For example, we found that it is much more efficient to process a GOB at a time than to process MBs one by one. The main reason is that those computational intensive functions, e.g. IDCT shown in Fig. 3.5, can be called repeatedly during GOB processing to reduce the program cache miss rate. Our simulation results show that the performance difference for the average cases can be as large as 294%.

The simulation speed of SDFS is sufficiently fast for experiments that are quite typical in practical design chores. On a Dell D600 laptop (1.5GHz Pentium4 CPU + 512MB DDR SDRAM), it takes about 130ms to simulate decoding 1 MB,

which equally means that it takes about 4.28×10^4 host cycles to simulate a target cycle. Compared to [82] which couples an ISS with a VHDL simulator and takes about 1.5×10^6 host cycles to simulate a target cycle, SDFS is about 2 orders of magnitude faster but achieves comparable accuracy. Since SDFS does not rely on any specific trace file as input, we found that simulating 10 frames is usually enough to estimate the system performance. Even for resolutions as large as D1, it only takes 3-4 minutes.

3.6 Conclusion

SDFS is presented for system wide performance estimation of multimedia applications. It carries out the simulation at a sufficiently low level to catch the detailed activities on the HW. Each block is completely parameter-driven so that the system architect does not need to write any code but to focus on building the application CDFG and HW graph as the simulator input. We demonstrate the usefulness of SDFS with real application examples from industry. SDFS takes about 4.28×10^4 host cycles to simulate a target cycle when the simulation is carried on a Dell D600 laptop (1.5GHz Pentium4 CPU + 512MB DDR SDRAM). Compared to [82] which couples an ISS with a VHDL simulator and takes about 1.5×10^6 host cycles to simulate a target cycle, SDFS is about 2 orders of magnitude faster but achieves comparable accuracy. Since SDFS does not depend on any specific video stream as input, simulating 10 frames is quite sufficient for architecture evaluation. Even for resolutions as large as D1, a simulation only takes 3-4 minutes. The simulator has demonstrated reliable performance estimation capability for use in system-level tradeoff analysis. The error of the best and average case estimation result is within 6% for our experiments; and the error of worst case result is within 13%. This applies even to changes in application CDFG and in HW.

The $func$ parameters in the application CDFG can be obtained by profiling

similar legacy applications or from benchmark results that are only valid under certain execution conditions. Compared to other research work, our tool is truly able to estimate the system performance corresponding to changes in execution conditions brought about by different *func* composition and/or HW architecture. The desired information from a simulation can be collected by inserting probepoints at appropriate places. The information thus collected not only helps to generate a design specification with high confidence, but may also suggest how to optimize the implementation solution.

Currently, *func*s need to be inserted at appropriate places in the application CDFG to model the cache activities, which requires expertise knowledge of the SW implementation. We plan to improve the simulator to automate cache modeling.

Chapter 4

Real-Time Simulation Platform

4.1 Introduction

Many embedded systems combine digital signal processing (DSP) functions with SW-implemented controllers. Because of the high cost of HW development, it is critical to reuse as much of the HW core as possible, and to meet real-time performance requirements by redesigning the on-chip HW accelerators, DMAs and bus architectures for the specific application of interest. To realize this strategy, tools are required to validate design decisions at the system level, and this is made difficult by the practical need to develop the application SW in parallel with the HW. The research challenge is to invent a simulation platform for the application SW that can provide sufficient precision to guarantee performance, is sufficiently fast, and be compatible with the SW development environment. From the industry perspective, it is extremely difficult to convince SW engineers to change their own development environment [137].

The real-time simulation platform (RTSP) presented here is implemented on legacy DSPs. To the best of our knowledge, it is the first simulator that truly enables HW/SW co-development by offering the same SW development environment as if

the rhw were available. RTSP applies the concept of Real-Time Virtual Machines [132] to the domain of simulation. Application SW executes directly on legacy DSPs to eliminate the need for instruction set simulation (ISS). Simulation accuracy is achievable since the DSP core remains the same in our approach. For each rhw, a corresponding $simcom$ is constructed running on a legacy DSP to simulate its behavior. Each $simcom$ shares an appropriate fraction of the chosen DSP so that the simulation is carried out at a proportional speed of the rhw. Revision to the HW design specification only results in changing the $simcom$'s SW behavior model which is a lot easier than changing the RTL model. More importantly, changes affecting the speed of a rhw only requires changing the sharing rate of $simcom$ which does not affect the application SW and the $simcom$ itself. This platform is affordable since legacy DSP evaluation version module (EVM) boards can be reused for the simulation purpose.

For each legacy DSP, scheduling a set of $simcom$(s) on it resembles the traditional periodic scheduling problem which has been well addressed. For a $simcom$ with share rate α, such a scheduler guarantees that $\forall\ t$, the difference between the actual supply the $simcom$ has received and the normal supply it is supposed to receive is bounded. However, periodic schedulers do not determine how much the $simcom$ can use the given supply and thus still cannot guarantee that all $simcom$s progress at relatively the same speed. The problem is illustrated in Fig. 4.1.

Fig. 4.1-(a) shows the execution scenario of rhw_1 and rhw_2. Suppose at t_1, rhw_1 sends to rhw_2 an event ε_1 to trigger a job on rhw_2 and then waits until it finishing. rhw_2 finishes the job at t_2 and sends event ε_2 back to rhw_1 to allow it continue. Assuming the speed ratio between each pair of rhw and $simcom$ is γ, Fig. 4.1-(b) illustrates the ideal simulation that ε_1 and ε_2 are simulated at $\gamma \cdot t_1$ and $\gamma \cdot t_2$, respectively. Actual simulation cannot guarantee that since the scheduling quantum is in discrete size. A bad case is that ε_1 is delayed by d_1 and ε_2 can further be delayed by d_2 as shown in Fig. 4.1-(c). It is obvious that the delay can be

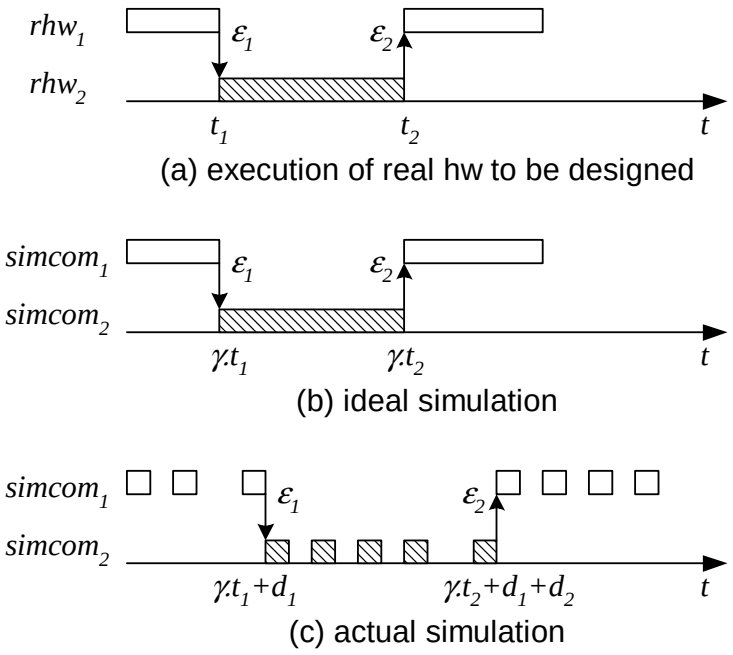

Figure 4.1: Simulation Problem

accumulated to make it unbounded and thus simulation fidelity is violated.

Emitting an event too early is also a problem but it can be solved by buffering early events until the time they can be delivered. In this chapter, we address the problem how to make a *simcom* "catch up" if some jobs start late. A two-level scheduler is designed for each legacy DSP to bound the delay. The simulation speed is intentionally slowed by a factor of S so that $\forall$ $simcom_i$ with rate α_i, it uses the supply at the actual rate $\frac{\alpha_i}{S}$. However, the chance for $simcom_i$ to get the supply is $\frac{K_i \cdot \alpha_i}{S}$ where $K_i \in \mathbb{N}$. The exact values of S and K_i are determined by the application's characteristics. The first level scheduler is a real-time periodic scheduler assigning quantums. The second level scheduler determines when a *simcom* is allowed to use the assigned quantum. The result is that when a *simcom*'s job is started late because of a delayed event, it still can "catch up" before finishing the job.

An audio and a video application are selected as real DSP industry applications for experiment. The results show that the scheduling overhead is negligible, the simulation speed is fast, and the simulation accuracy is accurate.

The rest of the chapter is organized as following. Section 4.2 gives a summary on related work. Section 4.3 explains the scheduling algorithm, focusing on the second level scheduler. Section 4.4 explains how to find appropriate K_i value for $simcom_i$ and S. Section 4.5 provides a heuristic algorithm to assign $simcom$s to the available $simhw$s. Some important implementation features are described in section 4.6. Section 4.7 shows the experiment results and section 4.8 draws the conclusion.

4.2 Related Work

The methodology presented in this chapter is motivated from [133] in which a real-time virtual processor prototype is implemented based on enhanced Linux kernel. An application on a virtual processor is guaranteed to get its share and cleanly isolated from other processes. RTSP presented in this paper is implemented on TI DSPs based on DSP/BIOS [15].

Although theoretically many real-time periodic schedulers can be used as the first level scheduler, it is desirable to have the most "fair" scheduler so that each $simcom$ has a fair chance to get quantums. The chairman assignment algorithm [162] proposed by Tijdeman has been proved to be optimal in the sense that $\forall\ t$, the deviation between the actual and normal supply for a $simcom$ is bounded and the bound is the smallest among all algorithms. Stoica in [157] proposed an efficient floating point implementation scheme. In this chapter, we propose the fixed point implementation which is friendlier to DSP platform.

Numerous techniques have been published for performance estimation. Static abstract modeling approaches are proposed in [151] [116] to support early architecture level DSE. [143] presented a behavior model implemented using SystemC [6].

Each block is assumed to consist of multiple sections each of which has a static execution cost. [53] proposed an estimation approach by analyzing the application source code and generating a performance profile. Such approaches are not able to estimate the performance by simulating the actual SW to be implemented. In contrast, the performance data of RTSP is obtained by directly executing the SW on legacy DSPs with similar architecture as the *rhw*, and thus the result is more accurate. Coupling an ISS with a RTL simulator modeling the *rhw*, i.e. [115] [82], or using the prototyping HW such as [58], allows executing the actual SW code to get accurate simulation result. However, constructing such high-fidelity platforms is labor intensive and expensive. More importantly, they are typically very slow and have limited debugging capability which prohibits any complex SW to be developed on them [169]. In contrast, RTSP exploits legacy HW and therefore is much cheaper and by applying the behavior model to eliminate ISS, RTSP is also fast.

To the best of the author's knowledge, [137] is the only published work trying to create a simulation engine that is compatible with the SW development environment in HW/SW co-design. It requires two host computers. The application SW is simulated on the first one. All the *simcom*s are constructed in SystemC and VHDL and simulated on the other one. The communication between the two hosts is socket-based so that simulating a simple register-read can take as long as 3.7 ms. The synchronization problem between HW and SW is not addressed. Compared to [137], RTSP achieves higher simulation accuracy by deploying the legacy DSPs as the *simhw* and handling synchronization between *simcom*s properly. RTSP also achieves much higher simulation speed by utilizing Serial RapidIO (SRIO) [12] as the much more efficient communication link between *simhw*s. The bandwidth of SRIO is 20Gb/s bidirectionally.

[98] presented a tool to simulate video application's dataflow in low level without ISS. It can be used with RTSP to complement each other. The information collected from RTSP can be fed to this tool to refine HW design specification which

in turn may suggest changes to the behavior models and parameters in RTSP. [172] [96] showed the effectiveness of combining RTOS model with the application SW to improve simulation accuracy. RTSP is implemented to allow integration of these models.

Simulation speed can be severely downgraded by frequent synchronization between *simcom*s. [108] proposed the optimized conservative approach in which synchronization point is predicted by SW analysis. [96] provides an algorithm for the multi-task context. The effectiveness of such approaches directly depends on knowledge to the timing characteristics of the simulation SW, and the insertion of prediction code is error-prone. [43] tried to reduce the IPC frequency by resorting to compile-time/run-time scheduling. All these approaches do not remove unnecessary synchronization completely. [113] [172] introduced a technique called virtual synchronization which combines the event-driven scheduler with data-driven model to remove unnecessary synchronization under the assumption that the output of the simulation SW only depends on event ordering but not the arrival time of each event. RTSP does not need this assumption since the scheduler is designed to force each *simcom* to progress together modulo some bounded jitter.

4.3 Algorithm

4.3.1 Annotations

The following are the notations to assist describing the scheduling algorithm.

Notations 4.3.1 *$\gamma = R \cdot S$ is the ratio between the speed of $\mathbb{H}$ and simulation, where R is the minimal achievable ratio determined by $\mathbb{SH}$ without considering bounding delay, and S is the slowdown factor to bound delay.*

Notations 4.3.2 *α_i and K_i are $simcom_i$'s scheduling parameters. $simcom_i$ progresses at rate $\frac{\alpha_i}{S}$ but receives supply from its simhw at rate $= \frac{\alpha_i \cdot K_i}{S}$ ($K_i \in \mathbb{N}$ and*

$K_i \geq 1$).

Notations 4.3.3 *Q is the scheduling quantum determined by the scheduler on a simhw. Q_i denotes the quantum for $simcom_i$. $\forall\ simcom_i \in simhw_j$ and $simcom_k \in simhw_j$ $(i \neq k)$, $Q_i = Q_k$.*

Notations 4.3.4 *L_i is the difference between the actual and normal supply of $simcom_i$ and is determined by the first level scheduler on $simhw_j$ ($simcom_i \in simhw_j$). $\forall\ t$, it is assumed that*

$$|s_i(t) - t \cdot \frac{K_i \cdot \alpha_i}{S}| \leq L_i \cdot Q_i \tag{4.1}$$

where $s_i(t)$ and $t \cdot \frac{K_i \cdot \alpha_i}{S}$ are the actual and normal supply function of $simcom_i$. $\forall$ $simcom_i \in simhw_j$ and $simcom_k \in simhw_j$ $(i \neq k)$, $L_i = L_k$.

Notations 4.3.5 *$V_i(t)$ is the virtual time of $simcom_i$. $\forall\ t$ in the actual simulation, $V_i(t)$ is the time $simcom_i$ would have reached in an ideal simulation.*

Notations 4.3.6 *$\mathbb{C}_i$, $c_i^k = \{p_i^k, \mathbb{E}_i^k, t\varepsilon_i^k, \langle ts_i^k, Vs_i^k\rangle, \langle te_i^k, Ve_i^k\rangle\}$:*

$\forall\ simcom_i$, $\mathbb{C}_i$ is the set of jobs to be simulated. The number of jobs is $|\mathbb{C}_i|$ and c_i^k is the k^{th} job. At the end of c_i^k, $simcom_i$ sends an event to itself to trigger c_i^{k+1} if $k < |\mathbb{C}_i|$. Event(s) may also be sent to other simcoms to trigger their job(s).

- *ts_i^k (te_i^k): actual start (finishing) time of c_i^k.*
- *Vs_i^k (Ve_i^k): virtual start (finishing) time of c_i^k.*
- *p_i^k: length of c_i^k.*
- *$\mathbb{E}_i^k$: the set of event(s) $simcom_i$ needs to receive before c_i^k starts. $\varepsilon_j^l \in \mathbb{E}_i^k$ means the the event is sent by $simcom_j$ at the end of c_j^l.*
- *$t\varepsilon_i^k$: the time that the last event in $\mathbb{E}_i^k$ is received during simulation.*

Notations 4.3.7 *Θ_i is the set of simcom(s) that send event(s) to $simcom_i$ to trigger its job(s). That is, $\Theta_i = \{\mathbb{E}_i^k | k = 1...|\mathbb{C}_i|\}$*

Notations 4.3.8 *$\triangle_i = \frac{2 \cdot L_i \cdot Q_i \cdot S}{K_i \cdot \alpha_i}$ is the bound to be established for $simcom_i$ such that $\forall\ c_i^k$, $te_i^k - Ve_i^k \leq \triangle_i$.*

Notations 4.3.9 *$state_i(t)$ is the state of $simcom_i$ at time t during simulation. It can either be $exec_i^k$ meaning that $simcom_i$ is simulating c_i^k, or $idle_i^k$ meaning that $simcom_i$ is waiting for c_i^k to start.*

Notations 4.3.10 *$t \rightarrow i$ / $\overline{t \rightarrow i}$ denotes that a Q with interval $[t, t + Q_i]$ is / is not assigned to $simcom_i$.*

Notations 4.3.11 *$t \rightrightarrows i$ / $\overline{t \rightrightarrows i}$ denotes that $simcom_i$ is / is not allowed to use the Q assigned in $[t, t + Q_i]$.*

Notations 4.3.12 *$u_i[t_0, t_1]$ denotes the interval used by $simcom_i$ within a quantum Q allocated to it where Q spans $[t_0, t_0 + Q_i]$. Note that $simcom_i$ may actually use only a portion of Q.*

4.3.2 Scheduling Algorithm

We assume that it is always possible to find R so that $\forall\ simcom_i$, $\frac{speed(rhw_i)}{speed(simcom_i)} = R$ by assigning $simcom_i$ an appropriate rate α_i. R determines the fastest achievable simulation speed without considering bounding the delay. The following example shows how to find α_i and R.

Example 4.3.1 *$|\mathbb{H}| = 3$, $|\mathbb{SH}| = 2$, $simcom_1 \in simhw_1$, $simcom_2 \in simhw_1$, and $simcom_3 \in simhw_2$.*

$$\frac{speed(rhw_1)}{speed(simcom_1)} = 2.5; \quad \frac{speed(rhw_2)}{speed(simcom_2)} = 2; \quad \frac{speed(rhw_3)}{speed(simcom_3)} = 4;$$

To make $\frac{speed(rhw_1)}{speed(simcom_1)} = \frac{speed(rhw_2)}{speed(simcom_2)} = 2$, $\frac{\alpha_1}{\alpha_2} = \frac{2.5}{2}$.
Setting $\alpha_1 = \frac{2.5}{4.5}$, $\alpha_2 = \frac{2}{4.5}$ *makes* $\frac{speed(rhw_{1,2})}{speed(simcom_{1,2})} = 4.5$.
Setting $\alpha_3 = \frac{4}{4.5}$ *makes* $\frac{speed(rhw_3)}{speed(simcom_3)} = R = 4.5$.

Simulation is intentionally slowed down by S to bound delay. A two-level scheduler is employed in each $simhw$. $\forall\ simcom_i$, the first level periodic scheduler assigns Qs to $simcom_i$ at rate $\frac{\alpha_i \cdot K_i}{S}$. The second level scheduler forces $simcom_i$ to use the assigned Qs at rate $\frac{\alpha_i}{S}$. Fig. 4.2 illustrates the idea for an example when $|\mathbb{H}| = 2$, $R = 3$, $p_1^1 = p_2^1 = \frac{1}{3}$ and $\alpha_1 = \alpha_2 = \frac{1}{3}$.

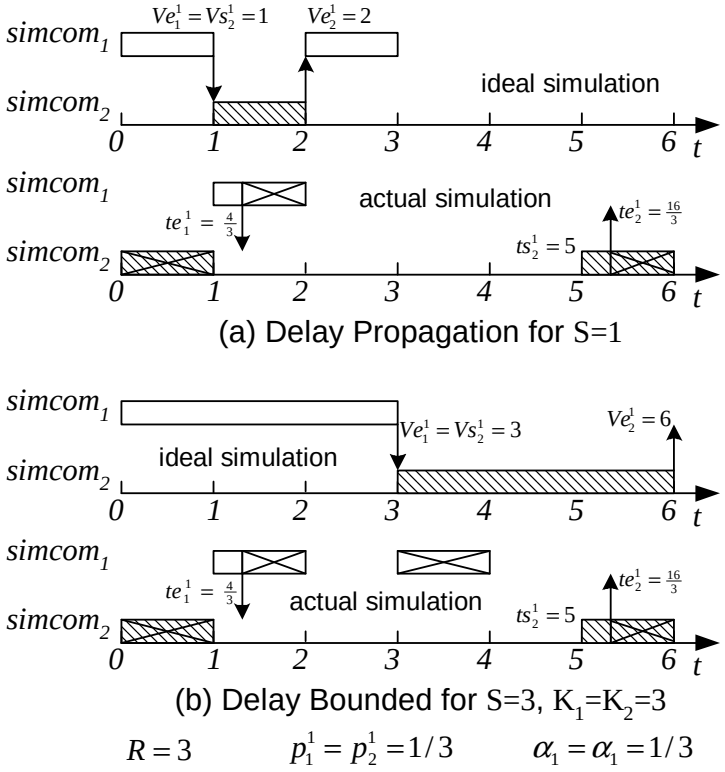

Figure 4.2: Idea to Bound Delay

Example 4.3.2 *If simulation is not slowed down ($S = 1$), $simcom_1$ would finish c_1^1 and trigger c_2^1 at $Ve_1^1 = R \cdot p_1^1 = 1$ in ideal simulation. $simcom_2$ would finish c_2^1 and*

trigger c_1^2 *at* $Ve_2^1 = 1 + R \cdot p_2^1 = 2$. *In the actual simulation,* $simcom_1$ *and* $simcom_2$ *can only receive one* Q *of every three because* $\alpha_1 = \alpha_2 = \frac{1}{3}$. *It could happen that* $simcom_2$ *receives the first* Q *but cannot use it, and* $simcom_1$ *receives the second one and finishes* c_1^1 *at* $te_1^1 = \frac{4}{3}$. *Suppose that* $simcom_2$ *would receive its next* Q *at* $t = 5$, c_2^1 *will be finished at* $te_2^1 = \frac{16}{3}$. *Clearly the delay is accumulated.*

Fig. 4.2-(b) shows the case when simulation is slowed by a factor of 3 *with both* K_1 *and* K_2 *being set to* 3. *Now* $Ve_1^1 = 3$ *and* $Ve_2^1 = 6$. *In the actual simulation, both simcoms still receive one* Q *of every three from the first level scheduler since* $\frac{K_1 \cdot \alpha_1}{S} = \frac{K_2 \cdot \alpha_2}{S} = \frac{1}{3}$, *but the second level scheduler allows the simcom to use it only if a job can be simulated at that moment. For example,* $simcom_2$ *does not use the given* Q *at* $t = 0$ *but uses the one at* $t = 5$ *when* c_2^1 *can be started. Delay is bounded in this case because* $te_1^1 < Ve_1^1$ *and* $te_2^1 < Ve_2^1$.

The first level scheduler can be any scheduler that guarantees Equ. (4.1). Tijdeman's chairman assignment algorithm is chosen since it has been proved to achieve the minimal L value among all algorithms. Fig. 4.3 summarizes the algorithm. $\forall\ simhw_j$, it is activated at every $t = m \cdot Q$ ($m \in \mathbb{N}$) assigning the next Q to an eligible *simcom* if there is any.

Fig. 4.4 shows the second level scheduler which consists of three portions: 1) *pre-scheduler*, 2) *post-scheduler*, and 3) *event-handler*. The pre-scheduler determines whether $simcom_i$ is allowed to use the given Q when $t_0 \rightarrow i$. $case_{1.1}$, $case_{1.2}$ and $case_{1.3}$ imply that $\forall\ simcom_i$, it is allowed to use a given Q only if it is in the *exec* state. The pre-scheduler also implies that $\forall\ c_i^k$, it always starts at the beginning of a given Q.

The post-scheduler updates $simcom_i$'s state and virtual time at $t_0 + Q_i$ when $t_0 \rightrightarrows i$ and $u_i[t_0, t_1]$. It may use part of the Q to finish the current job ($case_{2.1}$), or use the whole Q and then is forced to yield ($case_{2.2}$).

$n = |simhw_j|$, (number of $simcom$s on $simhw_j$)
initially ($t = 0$): $\forall\ simcom_i \in simhw_j$,
$L_i := \begin{cases} 1 - 0.5/(n-1); & \text{if } n > 1 \\ 0.5; & \text{otherwise} \end{cases}$
$lag_i(0) := 0$;

scheduling at $t = m \cdot Q$ ($m \in \mathbb{N}$):
$\forall\ simcom_i \in simhw_j$,
define $simcom_i$ is eligible if $lag_i(t) + L_i + \frac{K_i \cdot \alpha_i}{S} - 1 \geq 0$
define $d_i(t) := \frac{[L_i - lag_i(t)] \cdot S}{K_i \cdot \alpha_i}$ if $simcom_i$ is eligible
$lag_i(t) := lag_i(t) + \frac{K_i \cdot \alpha_i}{S}$;
if $\exists\ simcom_k$ that $d_k(t)$ is minimal among all eligible $simcom$s.
$t \rightarrow k$; (assign the next Q to $simcom_k$)
$lag_k(t) := lag_k(t) - 1$;
end if

Figure 4.3: Tijdeman's Algorithm

The event-handler determines the supposed start time of c_i^k, which is Vs_i^k for $simcom_i$. A variable $V\varepsilon_i^k$ is kept for c_i^k to record the time when the latest event is supposed to be received among all the event(s) to trigger c_i^k. When an event $\varepsilon_j^l \in \mathbb{E}_i^k$ is sent to $simcom_i$ by $simcom_j$ at the end of c_j^l (t_1), the time when it is supposed to be received ($V_j(t_1)$) is compared with $V\varepsilon_i^k$. $V\varepsilon_i^k$ is updated if $V_j(t_1) > V\varepsilon_i^k$. When all the events in $\mathbb{E}_i^k$ are received, Vs_i^k is known as $V\varepsilon_i^k$.

4.3.3 Correctness Proof of the Scheduler

The objective of this section is to prove that $\exists\ S$ and K_i for $simcom_i$ so that $\forall\ c_i^k$, $te_i^k - Ve_i^k$ is bounded. The proof can be divided to the following steps. Firstly we formally define the meaning of "catch up" if a job is started late. Theorem 4.3.1, corollary 4.3.1 and 4.3.2 prove the longest time $simcom_i$ needs to wait to start receiving the next Q. Then lemma 4.3.1 and 4.3.2 prove the biggest possible delay for $\forall\ c_i^k$ to start, using theorem 4.3.1, corollary 4.3.1 and 4.3.2. Given a bounded start delay and length of c_i^k, lemma 4.3.3, 4.3.4 and 4.3.5 prove that c_i^k can catch

pre-scheduler: when $t_0 \rightarrow i$
  $case_{1.1}$: if $state_i(t_0) = exec_i^k$ & $V_i(t_0) < t_0 + Q_i$
    $t_0 \rightrightarrows i$;
  end $case_{1.1}$

  $case_{1.2}$: if $state_i(t_0) = idle_i^k$, $\exists m \in \mathbb{N}$ that $t_0 \geq m \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i}$ and
  $Vs_i^{k+1} \in [m \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i}, (m+1) \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i})$
    $state_i(t_0) := exec_i^k$; $V_i(t) := Vs_i^k$; $ts_i^k := t_0$; $t_0 \rightrightarrows i$;
  end $case_{1.2}$

  $case_{1.3}$: if neither $case_{1.1}$ nor $case_{1.2}$ applies
    $\overline{t_0 \rightrightarrows i}$;
  end $case_{1.3}$
end *pre-scheduler*

post-scheduler: at $t_0 + Q_i$ when $t_0 \rightarrow i$, $u_i[t_0, t_1]$ and $state_i = exec_i^k$
  $case_{2.1}$: if $t_1 < te_i^k$
    $V_i(t_1) := V_i(t_0) + (t_1 - t_0) \cdot \frac{S}{\alpha_i}$
  end $case_{2.1}$

  $case_{2.2}$: if $t_1 = te_i^k$
    $state_i(t_1) := idle_i^{k+1}$;
    $V_i(t_1) := V_i(t_0) + (t_1 - t_0) \cdot \frac{S}{\alpha_i}$
  end $case_{2.2}$
end *post-scheduler*

event-handler: when $\varepsilon_j^l \in \mathbb{E}_i^k$ is received at t_1
  define $V\varepsilon_i^k := 0$ initially
  if $V_j(t_1) > V\varepsilon_i^k$, $V\varepsilon_i^k := V_j(t_1)$
  if all events in $\mathbb{E}_i^k$ have been received, $Vs_i^k := V\varepsilon_i^k$;
end *event-scheduler*

Figure 4.4: Second Level Scheduler

up c_i^k. Lemma 4.3.6 proves that when c_i^k has been caught up by $simcom_i$, its finish time will be delayed at most by $\triangle_i = \frac{2 \cdot L_i \cdot Q_i \cdot S}{\alpha_i \cdot K_i}$. Finally theorem 4.3.2 proves that $\exists\ S$ and K_i for each $simcom_i$ that $\forall\ c_i^k$, $te_i^k - Ve_i^k \leq \triangle_i$, using previously proved results.

Definition 4.3.1 *"catch up": $\forall\ c_i^k$ to be simulated, $simcom_i$ is defined to have caught up (when c_i^k was started late) if $\exists\ t \in [ts_i^k, te_i^k]$ that $t = V_i(t)$.*

Theorem 4.3.1 The Tijdeman's algorithm guarantees that $\forall\ simcom_i, \forall\ t, |lag_i(t)| = |t \cdot \frac{K_i \cdot \alpha_i}{S} - s_i(t)| \leq L_i \cdot Q_i$, where L_i is defined by Equ. (4.2) and L_i is the minimal among all algorithms.

$$L_i = \begin{cases} 1 - \frac{0.5}{n-1} & \text{if } n > 1 \\ 0.5 & \text{otherwise} \end{cases} \tag{4.2}$$

where n is the number of $simcom$(s) on $simhw_j$ including $simcom_i$.

Proof: The proof can be found in [162]. ■

Corollary 4.3.1 $\forall\ simcom_i$, if $\overline{t_0 \rightarrow i}$, $t_1 \rightarrow i$, $s_i(t_1) - s_i(t_0) = n \cdot Q_i\ (n \in \mathbb{N})$, then $t_1 - t_0 \leq (2 \cdot L_i + n) \cdot \frac{Q_i \cdot S_i}{\alpha_i \cdot K_i} = \triangle_i + n \cdot \frac{Q_i \cdot S_i}{\alpha_i \cdot K_i}$.

Proof:

$$\text{Theorem 4.3.1} \Rightarrow \quad s_i(t_0) - t_0 \cdot \frac{K_i \cdot \alpha_i}{S} \leq L_i \cdot Q_i \quad (a)$$

$$\frac{K_i \cdot \alpha_i}{S} \cdot t_1 - s_i(t_1) \leq L_i \cdot Q_i \quad (b)$$

$$(a) + (b) \Rightarrow t_1 - t_0 \leq (2 \cdot L_i + n) \cdot \frac{Q_i \cdot S}{\alpha_i \cdot K_i}$$

■

Corollary 4.3.2 $\forall\ simcom_i$, if $t_0 \rightarrow i$, $t_2 \rightarrow i$, $t_1 \in [t_0, t_0 + Q_i)$, $s_i(t_2) - s_i(t_0) = n \cdot Q_i$, $(n \geq 1,\ n \in \mathbb{N})$, then $t_2 - t_1 \leq \triangle_i + Q_i + (n-1) \cdot \frac{Q_i \cdot S_i}{\alpha_i \cdot K_i}$

Proof: Corollary 4.3.1 $\Rightarrow t_2 - (t_0 + Q_i) \leq \triangle_i + (n-1) \cdot \frac{Q_i \cdot S}{\alpha_i \cdot K_i}$

$$t_1 \geq t_0 \Rightarrow t_2 - t_1 \leq \triangle_i + Q_i + (n-1) \cdot \frac{Q_i \cdot S}{\alpha_i \cdot K_i}$$

■

Lemma 4.3.1 shows the biggest delay for c_i^k to start if no triggering event is received late.

Lemma 4.3.1 $\forall\ c_i^k$, if $t\varepsilon_i^k \leq Vs_i^k$, then $ts_i^k - Vs_i^k \leq \triangle_i + Q_i$.

Proof: At $t\varepsilon_i^k$, $state_i(t\varepsilon_i^k) = idle_i^k$. c_i^k starts later when $case_{1.2}$ in Fig. 4.4 applies. That is, $\exists\ m \in \mathbb{N}$ that $ts_i^k \geq m \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i}$, $m \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i} \leq Vs_i^k < (m+1) \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i}$.

The proof covers the following two conditions.

1) If $\exists\ t_0$ that $t_0 \rightarrow i$ and $m \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i} \in [t_0, t_0 + Q_i]$

Corollary 4.3.2 $\Rightarrow \exists\ t_1$, $t_1 \rightarrow i$, $s_i(t_1) - s_i(t_0) = Q_i$ and $t_1 \leq \triangle_i + Q_i + m \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i}$.
Since $case_{1.2}$ can be applied at t_1, $ts_i^k = t_1 \leq \triangle_i + Q_i + m \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i}$.

2) If $\nexists\ t_0$ that $t_0 \rightarrow i$ and $m \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i} \in [t_0, t_0 + Q_i]$

Corollary 4.3.1 $\Rightarrow \exists\ t_1 > t_0$, $t_1 \rightarrow i$, $s_i(t_1) = s_i(t_0)$ and $t_1 \leq \triangle_i + m \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i}$.
Since $case_{1.2}$ can be applied at t_1, $ts_i^k = t_1 \leq \triangle_i + m \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i}$.

$Vs_i^k \geq m \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i} \Rightarrow ts_i^k - Vs_i^k \leq \triangle_i + Q_i$.

■

Lemma 4.3.2 shows the start delay bound for c_i^k if the biggest delay of all the triggering event(s) is d.

Lemma 4.3.2 $\forall\ c_i^k$, if $t\varepsilon_i^k > Vs_i^k$ and $t\varepsilon_i^k - Vs_i^k = d$, then $ts_i^k - Vs_i^k \leq d + \triangle_i + Q_i$.

Proof: It can be proved similarly as lemma 4.3.1. ■

Lemma 4.3.3 derives the condition for c_i^k to catch up when it is started late by d and the length is $\leq Q_i$.

Lemma 4.3.3 $\forall\ c_i^k$, if $ts_i^k - Vs_i^k = d > 0$, $p_i^k \leq Q_i$ and $S \geq \alpha_i \cdot (\frac{d}{p_i^k} + 1)$, then $\exists\ t_1 \in (ts_i^k, te_i^k]$, $V_i(t_1) = t_1$.

Proof:

$$p_i^k \le Q_i \Rightarrow te_i^k = ts_i^k + p_i^k = Vs_i^k + d + p_i^k$$
$$S \ge \alpha_i \cdot (\frac{d}{p_i^k} + 1) \Rightarrow \frac{p_i^k \cdot S}{\alpha_i} \ge p_i^k + d$$
$$Ve_i^k = Vs_i^k + \frac{p_i^k \cdot S}{\alpha_i} \Rightarrow Ve_i^k \ge Vs_i^k + p_i^k + d = te_i^k$$
$$ts_i^k > Vs_i^k, te_i^k \le Ve_i^k \Rightarrow \exists t_1 \in (ts_i^k, te_i^k], V_i(t_1) = t_1$$

■

Lemma 4.3.4 derives the condition for c_i^k to catch up when it is started late by d and the length is $> Q_i$.

Lemma 4.3.4 $\forall\, c_i^k$, if $ts_i^k - Vs_i^k = d > 0$, $p_i^k > Q_i$ and $S \cdot \frac{n+r}{\alpha_i} - \frac{S}{K_i} \cdot \frac{2 \cdot L_i + n - 1}{\alpha_i} - r - 1 \ge \frac{d}{Q_i}$, where $n = \lfloor \frac{p_i^k}{Q_i} \rfloor$, $r = \frac{p_i^k}{Q_i} - n$, then $\exists\ t_1 \in (ts_i^k, te_i^k]$, $V_i(t_1) = t_1$.

Proof:

$$\text{Corollary 4.3.2} \Rightarrow te_i^k \le ts_i^k + \triangle_i + (n-1) \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i} + Q_i + r \cdot Q_i$$
$$= Vs_i^k + d + (2 \cdot L_i + n - 1) \cdot \frac{S \cdot Q_i}{\alpha_i \cdot K_i} + (1 + r) \cdot Q_i \qquad (a)$$
$$Ve_i^k = Vs_i^k + (n + r) \cdot \frac{S \cdot Q_i}{\alpha_i} \qquad (b)$$
$$\frac{(b) - (a)}{Q_i} = S \cdot \frac{n+r}{\alpha_i} - \frac{S}{K_i} \cdot (\frac{2 \cdot L + n - 1}{\alpha_i}) - 1 - r - \frac{d}{Q_i} \qquad (c)$$
$$(c) \ge 0 \Rightarrow Ve_i^k \ge te_i^k$$
$$ts_i^k > Vs_i^k, te_i^k \le Ve_i^k \Rightarrow \exists t_1 \in (ts_i^k, te_i^k], V_i(t_1) = t_1$$

■

In reality, probably p_i^k can only be estimated so that r in lemma 4.3.4 is unknown. Assuming $r = 0$, lemma 4.3.5 can be applied.

Lemma 4.3.5 $\forall\, c_i^k$, if $ts_i^k - Vs_i^k = d > 0$, $p_i^k > Q_i$ and $S \cdot \frac{n}{\alpha_i} - \frac{S}{K_i} \cdot \frac{2 \cdot L_i + n - 1}{\alpha_i} - 1 \ge \frac{d}{Q_i}$, where $n = \lfloor \frac{p_i^k}{Q_i} \rfloor$, then $\exists\ t_1 \in (ts_i^k, te_i^k]$, $V_i(t_1) = t_1$.

Proof: the proof is similar to that of lemma 4.3.4.

■

Lemma 4.3.6 proves that $\forall\, c_i^k$ if it has been caught up (even though it started late), its finish delay is $\le \triangle_i$.

Lemma 4.3.6 $\forall\ c_i^k$, if $\exists\ t_0 \in [ts_i^k, te_i^k]$, $V_i(t_0) = t_0$, then $te_i^k \leq Ve_i^k + \triangle_i$.

Proof: Assuming t_0 is the latest time in $[ts_i^k, te_i^k]$ that $V_i(t_0) = t_0$, that is, $\nexists t_1 \in (t_0, te_i^k], V_i(t_1) = t_1$, the proof is divided into the following three conditions.

1) if $\nexists\ t^{'}, t^{'} \rightarrow i,\ t_0 \in [t^{'}, t^{'} + Q_i]$

$case_{1.1}$ is applied at $t_0 \Rightarrow \forall\ t \in (t_0, te_i^k]$, if $t \rightarrow i$, then $t \rightrightarrows i$. It means that $simcom_i$ will use every given Q after t_0 until c_i^k is finished.

Assuming $s_i(te_i^k) - s_i(t_0) = (n + r) \cdot Q_i$, $(n \in \mathbb{N},\ r \in \mathbb{R},\ n = \lfloor \frac{s_i(te_i^k) - s_i(t_0)}{Q_i} \rfloor)$,
corollary 4.3.1 $\Rightarrow te_i^k \leq t_0 + r \cdot Q_i + n \cdot \frac{Q_i \cdot S}{\alpha_i \cdot K_i} + \triangle_i$

$Ve_i^k = V_i(t_0) + (n + r) \cdot \frac{Q_i \cdot S}{\alpha_i} = t_0 + (n + r) \cdot \frac{Q_i \cdot S}{\alpha_i}$

$te_i^k - Ve_i^k \leq \triangle_i + [\frac{n \cdot S}{\alpha_i} \cdot (\frac{1}{K_i} - 1) + r \cdot (1 - \frac{S}{\alpha_i})] \cdot Q_i \leq \triangle_i$

2) if $\exists\ t^{'}, t^{'} \rightarrow i,\ t_0 \in [t^{'}, t^{'} + Q_i]$ and $t^{'} \rightrightarrows i$

$V_i(t_0) = t_0 \Rightarrow V_i(t_0 + Q_i) \geq t_0 + Q_i$

$\nexists\ t \in (t_0, te_i^k]$ that $V_i(t) = t \Rightarrow \forall\ t \in (t_0 + Q_i, te_i^k],\ V_i(t) \geq t \Rightarrow te_i^k \leq Ve_i^k$

3) if $\exists\ t^{'}, t^{'} \rightarrow i,\ t_0 \in [t^{'}, t^{'} + Q_i]$ but $\overline{t^{'} \rightrightarrows i}$

$case_{1.1}$ cannot be applied at $t^{'} \Rightarrow V_i(t^{'}) \geq t^{'} + Q_i$

$V_i(t^{'}) = V_i(t_0),\ t_0 = V_i(t_0) \Rightarrow V_i(t_0) = t_0 \geq t^{'} + Q_i$

$t_0 \in [t^{'}, t^{'} + Q_i] \Rightarrow t_0 \leq t^{'} + Q_i \Rightarrow t_0 = t^{'} + Q_i$

$case_{1.1}$ can be applied at $t_0 \Rightarrow \forall\ t \in (t_0, te_i^k]$, if $t \rightarrow i$, then $t \rightrightarrows i$.

Assuming $s_i(te_i^k) - s_i(t_0) = (n + r) \cdot Q_i$, $(n \in \mathbb{N},\ r \in \mathbb{R},\ n = \lfloor \frac{s_i(te_i^k) - s_i(t_0)}{Q_i} \rfloor)$,
corollary 4.3.1 $\Rightarrow te_i^k \leq t_0 + r \cdot Q_i + n \cdot \frac{Q_i \cdot S}{\alpha_i \cdot K_i} + \triangle_i$

$Ve_i^k = V_i(t_0) + (n + r) \cdot \frac{Q_i \cdot S}{\alpha_i} = t_0 + (n + r) \cdot \frac{Q_i \cdot S}{\alpha_i}$

$te_i^k - Ve_i^k \leq \triangle_i + [\frac{n \cdot S}{\alpha_i} \cdot (\frac{1}{K_i} - 1) + r \cdot (1 - \frac{S}{\alpha_i})] \cdot Q_i \leq \triangle_i$

■

Theorem 4.3.2 $\forall\ simcom_i$, define

$$P_i = \min\{p_i^k | 1 \leq i \leq |\mathbb{C}_i|\} \tag{4.3}$$

$$G_i = \begin{cases} 2 \cdot \alpha_i \cdot Q_i; & \text{if } P_i > Q_i \\ \alpha_i \cdot (P_i + Q_i); & \text{otherwise} \end{cases} \tag{4.4}$$

$$A_i = \begin{cases} \lfloor P_i/Q_i \rfloor \cdot Q_i; & \text{if } P_i > Q_i \\ P_i; & \text{otherwise} \end{cases} \tag{4.5}$$

$$f_i(K_i) = \begin{cases} \frac{(\lfloor P_i/Q_i \rfloor + 4 \cdot L_i - 1) \cdot Q_i}{K_i}; & \text{if } P_i > Q_i \\ \frac{2 \cdot L_i \cdot Q_i}{K_i}; & \text{otherwise} \end{cases} \tag{4.6}$$

$\forall\ simhw_j \in \mathbb{SH}$, define

$$S_j = \sum_{\in simhw_j}^{simcom_i} K_i \cdot \alpha_i \tag{4.7}$$

if $\forall\ simcom_i$, Equ. (4.8) and (4.9) are true, then $\forall\ c_i^k$, $te_i^k \leq Ve_i^k + \triangle_i$.

$$S \geq \frac{G_i}{A_i - f_i(K_i) - \alpha_i \cdot \max\{\frac{2 \cdot L_j \cdot Q_j}{K_j \cdot \alpha_j} | simcom_j \in \Theta_i\}} \tag{4.8}$$

$$S \geq \max\{S_j | simhw_j \in \mathbb{SH}\} \tag{4.9}$$

Proof: define

$$D_i = \triangle_i + Q_i + \max\{\triangle_j | simcom_j \in \Theta_i\} \tag{4.10}$$

From lemma 4.3.3, 4.3.5 and 4.3.6, we can prove that $\forall\ c_i^k$, if $ts_i^k - Vs_i^k \leq D_i$, then theorem 4.3.2 is true.

Next we prove that $\forall\ c_i^k$ and $\forall\ \varepsilon_j^l \in \mathbb{E}_i^k$ $(\mathbb{E}_i^k \neq \emptyset)$, if $te_j^l \leq Ve_j^l + \triangle_j$, then $ts_i^k \leq Vs_i^k + D_i$. The proof is divided to the following two conditions.

1) If $t\varepsilon_i^k \leq Vs_i^k$:

lemma 4.3.1 $\Rightarrow ts_i^k - Vs_i^k \leq \triangle_i < D_i$.

2) If $t\varepsilon_i^k > Vs_i^k$:

$\exists\ \varepsilon_m^l \in \mathbb{E}_i^k$ that $t\varepsilon_i^k = te_m^l \leq Ve_m^l + \triangle_m$

$Ve_m^l \leq Vs_i^k$ and $\triangle_m \leq \max\{\triangle_j | simcom_j \in \Theta_i\}$

$\Rightarrow t\varepsilon_i^k \leq Vs_i^k + \max\{\triangle_j | simcom_j \in \Theta_i\}$

Lemma 4.3.2 $\Rightarrow ts_i^k \leq t\varepsilon_i^k + \triangle_i + Q_i$

$\Rightarrow ts_i^k \leq Vs_i^k + \triangle_i + Q_i + \max\{\triangle_j | simcom_j \in \Theta_i\} \Rightarrow ts_i^k \leq Vs_i^k + D_i$.

Finally we prove that $\forall\ c_i^1$ with $\mathbb{E}_i^1 = \emptyset$, $te_i^1 \leq Ve_i^1 + \triangle_i$. $\mathbb{E}_i^1 = \emptyset \Rightarrow ts_i^1 = Vs_i^1 = 0$. Lemma 4.3.6 $\Rightarrow te_i^1 \leq Ve_i^1 + \triangle_i$. ■

$\forall\ c_i^k$, Equ. (4.10) defines its biggest start delay in which $\triangle_i + Q_i$ is caused by the scheduler for $simcom_i$ and $\max\{\triangle_j | simcom_j \in \Theta_i\}$ is introduced by the $simcom$s which will send event(s) to $simcom_i$ to trigger c_i^k.

4.4 Find K_i and S

The proof to theorem 4.3.2 shows that $\forall\ c_i^k$, its start time can be delayed at most by D_i as defined by Equ. (4.10) and its finish time can be delayed at most by $\triangle_i$, as long as Equ. (4.8) and (4.9) hold for $simcom_i$. This section solves the problem of finding S and an appropriate K_i for each $simcom_i$. It is desirable to find the minimal S since a bigger S means slower simulation speed. If the number of $simcoms$ to be considered are small and no α_i is too small, exhaustive searching can be performed by increasing K_i until Equ. (4.8) and (4.9) are met. Fig. 4.5, 4.6 and 4.7 provide a 3-step heuristic algorithm to quickly find the sub-optimal solution.

$\forall\ simcom_i$, function $\delta_i(j, K_j)$ is defined as the part of the job start delay introduced by the triggering event(s) that is sent by $simcom_j$, scaled by α_i/S.

$\forall\ simcom_i$, define function
$\delta_i(j, K_j) = \alpha_i \cdot \frac{2 \cdot L_j \cdot Q_j}{K_j \cdot \alpha_j}$, if $simcom_j \in \Theta_i$
$\theta_i(\Theta_i) = j$ if $\delta_i(j, K_j) = \max\{\delta_i(l, K_l) | simcom_l \in \Theta_i\}$

initially:
$\forall\ simcom_i,\ K_i := 1;$
$\forall\ simhw_j,\ S_j := \sum \alpha_i;\ (simcom_i \in simhw_j)$

step1:
for $i = 1 : |\mathbb{H}|$
  $j := \theta_i(\Theta_i);\ K_i := \max\{\lceil \frac{f_i(1)}{A_i} \rceil, \lceil \frac{\delta_i(j, K_j)}{A_i} \rceil\};$
end for

Figure 4.5: Find K_i and S: *Step1*

Function $\theta_i(\Theta_i)$ is the *simcom* which caused the maximum delay. *Step1* finds the initial value of K_i to start searching from for each $simcom_i$. It makes sure that $A_i \geq f_i(K_i)$ and $A_i \geq \delta_i(j, K_j)$ $(j = \theta_i(\Theta_i))$. Otherwise the denominator of Equ. (4.8) is always negative and Equ. (4.8) can never hold.

The objective of *Step2* is to make the denominator of Equ. (4.8) positive for each *simcom* by increasing the K parameters. For a particular $simcom_i$, the optimal solution might increase both K_i and K_l $(l = \theta_i(\Theta_i))$ by some amount but their exact values are unknown. The algorithm computes d_i (d_l) which is the amount to increase to make the denominator > 0 by only increasing K_i (K_l), and checks increasing which one will potentially make S increase less. If increasing K_i (K_l) is better, K_i (K_l) is increased by $\lceil \frac{d_i}{\omega} \rceil$ $(\lceil \frac{d_l}{\omega} \rceil)$ where ω is the adjustable step size to trade off search speed against accuracy. *Step2* exits if $\forall\ simcom_i$, the denominator of Equ. (4.8) is > 0.

Step3 finds the final solution. It is similar to *Step2*. The main difference is that df_i and df_l in *Step3* are obtained by solving the second-order equations while d_i and d_l in *Step2* are obtained by solving the linear equations.

for $i = 1 : |\mathbb{H}|$
start2:
  $l := \theta_i(\Theta_i)$;
  if $A_i - f_i(K_i) - \delta_i(l, K_l) < 0$
    $d_i := \lceil K_i \cdot [\frac{f_i(K_i)}{A_i - \delta_i(l,K_l)} - 1] \rceil$; $s_i := S_j + d_i \cdot \alpha_i$; $(simcom_i \in simhw_j)$
    $d_l := \lceil K_l \cdot [\frac{\delta_i(l,K_l)}{A_i - f_i(K_i)} - 1] \rceil$; $s_l := S_k + d_l \cdot \alpha_l$; $(simcom_l \in simhw_k)$
    if $s_i > s_l$
      $d_i := \lceil \frac{d_i}{\omega} \rceil$; $K_i := K_i + d_i$; $S_j := S_j + d_i \cdot \alpha_i$;
    else
      $d_l := \lceil \frac{d_l}{\omega} \rceil$; $K_l := K_l + d_i$; $S_k := S_k + d_l \cdot \alpha_l$;
    end if
    goto start2;
  end if
end for

Figure 4.6: Find K_i and S: *Step2*

4.5 Assign *simcom*s to $\mathbb{SH}$

4.5.1 Introduction

This section addresses the problem of assigning all the *simcom*s to the set of available legacy DSPs ($\mathbb{SH}$). If the number of *simcom*s and *simhw*s are small enough, an exhaustive searching method can be taken to find the optimal assignment solution. Otherwise, the heuristic algorithm in [95] can be applied to quickly find the suboptimal solution which is close to the optimal solution in most cases.

$\forall\ simhw_j$, the actual simulation time spent on it includes the following three portions: 1) computations made by all *simcom*(s) assigned to it, 2) scheduling overhead and 3) blocking cost due to serialized accessing to the SRIO link as common resource. It can be expressed by Equ. (4.11)-(4.14), where t_{j1}, t_{j2} and t_{j3} corresponds to portion 1), 2) and 3) respectively.

$$t_j = t_{j1} + t_{j2} + t_{j3} \tag{4.11}$$

for $i = 1 : |\mathbb{H}|$
start3:
  $l := \theta_i(\Theta_i)$; $S := \max\{S_j | simhw_j \in \mathbb{SH}\}$
  if $S < \frac{G_i}{A_i - f_i(K_i) - \delta_i(l, K_l)}$
    find df_i in $S_j + df_i \cdot \alpha_i = \frac{G_i}{A_i - f_i(K_i + df_i) - \delta_i(l, K_l)}$;
    $d_i := \lceil df_i \rceil$; $s_i := S_j + d_i \cdot \alpha_i$; $(simcom_i \in simhw_j)$
    find df_l in $S_k + df_l \cdot \alpha_l = \frac{G_i}{A_i - f_i(K_i) - \delta_i(l, K_l + df_l)}$;
    $d_l := \lceil df_l \rceil$; $s_l := S_k + d_l \cdot \alpha_l$; $(simcom_l \in simhw_k)$
    if $s_i > s_l$
      $d_i := \lceil \frac{d_i}{\omega} \rceil$; $K_i := K_i + d_i$; $S_j := S_j + d_i \cdot \alpha_i$;
    else
      $d_l := \lceil \frac{d_l}{\omega} \rceil$; $K_l := K_l + d_i$; $S_k := S_k + d_l \cdot \alpha_l$;
    end if
    goto start3;
  end if
end for

Figure 4.7: Find K_i and S: *Step3*

$$t_{j1} = \sum_{\in simhw_j}^{simcom_i} \sum_{k=1}^{|\mathbb{C}_i|} p_i^k \tag{4.12}$$

$$t_{j2} = cs \times \sum_{\in simhw_j}^{simcom_i} \times \lfloor \frac{t_{end}}{Q_j} \rfloor \tag{4.13}$$

$$t_{j3} = qc \times \sum_{\in simhw_j}^{simcom_i} |\mathbb{C}_i| \times [0.5 \times (|\mathbb{SH}| - 1) + 1] \tag{4.14}$$

Equ. (4.13) and Equ. (4.14) need some explanations. In Equ. (4.13), $\lfloor \frac{t_{end}}{Q_j} \rfloor$ is the total number of scheduler execution being made during simulation. The cost of each execution is proportional to the number of *simcom*s on $simhw_j$ ($\sum_{\in simhw_j}^{simcom_i}$), which is true for most practical scheduler implementation [167]. Using cs to denote the unit cost when there is only one *simcom* on $simhw_j$, the total scheduling overhead can be expressed by t_{j2}. In Equ. (4.14), qc represents the cost when each time a *simcom* exclusively utilizes the SRIO link to send event(s) to other *simcom*s when a job is done. Without considering the communication details

between all the *simcom*s, $qc \times \sum_{\in simhw_j}^{simcom_i} |\mathbb{C}_i|$ represents the total number of times that $simhw_j$ will use the SRIO link. Assuming the probability of obtaining the link is equal for all competing *simhw*s, it can be shown that each time the average blocking delay for $simhw_j$ to get the link can be approximated by $qc \times [0.5 \times (|\mathbb{SH}| - 1)]$ [95]. Therefore, t_{j3} represents the total cost that $simhw_j$ waits for and uses the SRIO link.

The objective function for the optimal assignment is expressed as Equ. (4.15), which balances the actual simulation cost among all *simhw*s.

$$\min \max_{\in \mathbb{SH}}^{simhw_j} t_j \tag{4.15}$$

4.5.2 Algorithm Finding Suboptimal Assignment Solution

Solving Equ. (4.15) is a binary integer programming problem. Norman in [138] showed that mapping parallel algorithms onto parallel architectures is an NP-hard problem in all but restricted cases. To obtain the optimal solution, an exhaustive search takes $\mathcal{O}(|\mathbb{SH}|^{|\mathbb{H}|})$ iterations. In case the number of *simcom*s and *simhw*s are not large, such an exhaustive search method can be taken. Otherwise, the heuristic algorithm needs to be applied to find the suboptimal solutions close to the optimal within significantly reduced time. Broadly speaking, most heuristic task assignment algorithms can be classified into three categories: *Local search algorithm, Genetic approaches and Greedy algorithms.* Our algorithm is similar to the greedy algorithm which starts from an empty solution set, repeatedly chooses a *simcom* based on certain selection strategy and assigns it to a *simhw* to obtain a partial solution. The algorithm continues until all the *simcom*s have been assigned. The key difference between our algorithm and the greedy one is that our algorithm keeps a bounded number of partial solutions after assigning a *simcom*. By adjusting the bound, it has the flexibility to trade off between the goodness of the obtained solution and the search time.

The following notations and defined to help present our algorithm.

Notations 4.5.1 *A partial solution is called a* $\langle n, m, k \rangle$ partial solution *if* n *simcoms have been assigned to* m *simhws among which* k *simhws are not assigned any simcom. We call those* k *simhws* idle *simhws. The other* $(m - k)$ *simhws are called* busy *simhws.*

Notations 4.5.2 *The most* promising $\langle n, m, k \rangle$ partial solution *among all* $\langle n, m, i \rangle$ *partial solutions is defined as the one which achieves the minimum of Equ. (4.15), only considering the* n *simcoms assigned and the* m *simhws.*

Notations 4.5.3 $\mathbb{B}_{\langle n,m,k \rangle}$ *is a set containing all the promising* $\langle n, m, k \rangle$ *partial solutions being kept. Its size is denoted as* $|\mathbb{B}_{\langle n,m,k \rangle}|$.

Notations 4.5.4 $\mathbb{A}_{\langle n,m,k \rangle}$ *is a set which containing all the* $\langle n, m, k \rangle$ *partial solutions derived from the partial solutions kept in the previous search step. Its size is denoted as* $|\mathbb{A}_{\langle n,m,k \rangle}|$.

When considering the assignment of the n^{th} *simcom*, it is easy to see that we only need to consider n *simhws* if $n < |\mathbb{SH}|$ since the other $(|\mathbb{SH}| - n)$ *simhws* will not be assigned any *simcom*. We will call them *backup simhws*.

Now let us look at how to obtain $\mathbb{A}_{\langle n,m,k \rangle}$, given all the promising partial solutions $\{\mathbb{B}_{\langle n-1,m',0 \rangle}, \ldots, \mathbb{B}_{\langle n-1,m',m'-1 \rangle}\}$ kept in the last search step, where

$$m' = \begin{cases} |\mathbb{SH}|; & \text{if } |\mathbb{SH}| < n - 1 \\ n - 1; & \text{otherwise} \end{cases} \tag{4.16}$$

The following two cases need to be considered.

1) If $|\mathbb{SH}| > n$,

   there is still backup *simhw*(s) available. By bringing in a backup *simhw* as $simhw_n$, the problem is to assign $simcom_n$ to any of the n *simhws*. By induction, we can show that

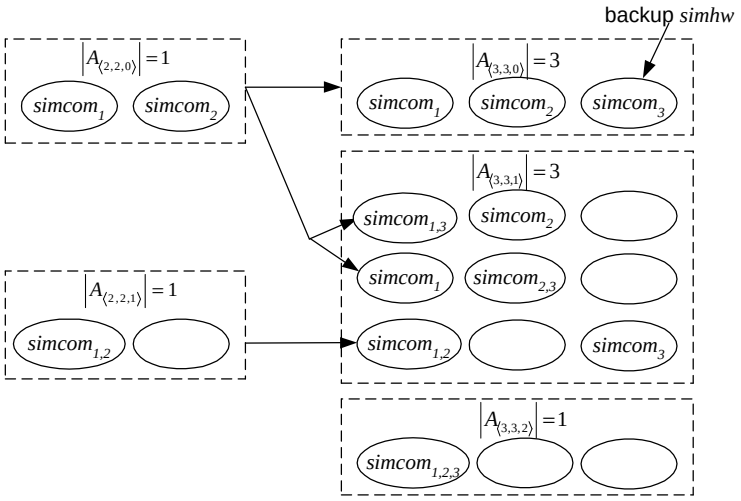

Figure 4.8: Assigning 3 *simcom*s to 3 *simhw*s

$$|\mathbb{A}_{\langle n,n,0\rangle}| = |\mathbb{B}_{\langle n-1,n-1,0\rangle}| = 1 \tag{4.17}$$

$$|\mathbb{A}_{\langle n,n,k\rangle}| = |\mathbb{B}_{\langle n-1,n-1,k\rangle}| + |\mathbb{B}_{\langle n-1,n-1,k-1\rangle}| \times (n-k) \quad (1 \leq k \leq n-1) \tag{4.18}$$

Equ. (4.17) is easy to understand because to assign n *simcom*s to n *simhw*s each of which has at least one *simcom*, there is only one possibility that each *simhw* gets exact one *simcom*.

To see how Equ. (4.18) is obtained, consider the first right hand side term $|\mathbb{B}_{\langle n-1,n-1,k\rangle}|$ which is the number of $\langle n-1, n-1, k\rangle$ partial solutions being kept. Each of them becomes a $\langle n, n, k\rangle$ partial solution if $simcom_n$ is assigned to $simhw_n$. Now consider the term $|\mathbb{B}_{\langle n-1,n-1,k-1\rangle}|$ which is the number of $\langle n-1, n-1, k-1\rangle$ partial solutions kept. Each of them can become a $\langle n, n, k\rangle$ partial solution if $simcom_n$ is assigned to any of the $(n-k)$ *busy* *simhw*s. Fig. 4.8 illustrates the idea when $n = 3$.

2) If $|\mathbb{SH}| \leq n$,

there is no backup *simhw* available, the problem is to assign $simcom_n$ to any of the existing $|\mathbb{SH}|$ *simhw*s. Similarly, we can show that

$$|\mathbb{A}_{\langle n,|\mathbb{S}\,\mathbb{H}|,k\rangle}| = |\mathbb{B}_{\langle n-1,|\mathbb{S}\,\mathbb{H}|,k\rangle}| \times (|\mathbb{SH}| - k) + |\mathbb{B}_{\langle n-1,|\mathbb{S}\,\mathbb{H}|,k+1\rangle}|$$
$$0 \leq k \leq |\mathbb{SH}| - 1 \tag{4.19}$$

The explanation above implies that the search space grows exponentially with $|\mathbb{SH}|$ and $|\mathbb{H}|$. To bound the search space, $|\mathbb{B}_{\langle n,m,k\rangle}|$ is set by Equ. (4.20). That is, at most C of the most promising $\langle n, m, k\rangle$ partial solutions are kept in each step. By adjusting C, our algorithm has the flexibility to trade off the goodness of the solution and the search time.

$$|\mathbb{B}_{\langle n,m,k\rangle}| = \min\{|\mathbb{A}_{\langle n,m,k\rangle}|, C\} \tag{4.20}$$

Selection of *simcom*s is in descending order of their effects to the objective function. The effect of a $simcom_i$ is defined in Equ. (4.21), where the first part of the effect is the total computation time and the second part is approximately the time utilizing the SRIO link. The larger the effect is, the more its assignment will affect the value of Equ. (4.15).

$$\sum_{k=1}^{|\mathbb{C}_i|}(p_i^k + qc) \tag{4.21}$$

The algorithm is summarized as following.

1. Select 2 *simcom*s and calculate $\mathbb{A}_{\langle 2,2,0\rangle}$ and $\mathbb{A}_{\langle 2,2,1\rangle}$.
2. If all $|\mathbb{H}|$ *simcom*s have been assigned, return the best solution. Otherwise go to step 3.
3. Select the next *simcom* denoted as $simcom_n$.
   (a) If there is a backup *simhw*, bring it in. $\forall\ k\ (0 \leq k \leq n-1)$, compute $\mathbb{A}_{\langle n,n,k\rangle}$ and keep the most promising $|\mathbb{B}_{\langle n,n,k\rangle}|$ partial solutions.

(b) Otherwise $\forall\ k\ (0 \leq k \leq |\mathbb{SH}| - 1)$, compute $\mathbb{A}_{\langle n,|\mathbb{S}\,\mathbb{H}|,k\rangle}$ and keep the most promising $|\mathbb{B}_{\langle n,|\mathbb{S}\,\mathbb{H}|,k\rangle}|$ partial solutions.

4. Go to step 3.

The number of search operations in step 3 is in the order of $\mathcal{O}(|\mathbb{SH}|^2 \cdot C)$ as implied by Equ. (4.18) and Equ. (4.19). Thus, the total number of search operations to find the suboptimal assignment solution is in the order of $\mathcal{O}(|\mathbb{H}| \cdot |\mathbb{SH}|^2 \cdot C)$.

4.6 Implementation

The two-level scheduler itself incurs simulation overhead and needs to be implemented as efficient as possible. Since in most cases fixed point DSPs will be used as *simhw*, the implementation should avoid floating point calculation and the division operation.

To implement the original Tijdeman's algorithm shown in Fig. 4.3, $\frac{K_i \cdot \alpha_i}{S}$ for each $simcom_i$ is converted to q_i which is in Q12 format. $\overline{q_i}$ is the inverse of q_i but also converted to Q12 format. The implementation is shown in Fig. 4.9.

$q_i := \frac{K_i \cdot \alpha_i}{S} \cdot 2^{b_i}$; $(2^{11} \leq q_i < 2^{12})$
$\overline{q_i} := \frac{S}{K_i \cdot \alpha_i} \cdot 2^{24-b_i}$;
$simcom_i$ is eligible if $lag_i(t) + (L_i - 1) \cdot 2^{b_i} + q_i \geq 0$
$d_i(t) := [L_i \cdot 2^{b_i} - lag_i(t)] \cdot \overline{q_i}$ for eligible $simcom_i$
if $t \rightarrow i$
  $lag_i(t) := lag_i(t) + q_i - 2^{b_i}$;
else
  $lag_i(t) := lag_i(t) + q_i$;
end if

Figure 4.9: Tijdeman's Algorithm Implementation

$\forall\ simcom_i$, $V_i(t)$ needs to be updated by the second level scheduler as shown in Fig. 4.4. Assuming $\frac{S}{\alpha_i} < 2^{16}$, it is converted to ρ_i in Q16 format. After $simcom_i$ using a Q in $[t_0, t_1]$ $(u_i[t_0, t_1])$, $V_i(t)$ can be updated as Equ. (4.22).

$$\begin{aligned} &\rho_i := \frac{S}{\alpha_i} \cdot 2^{\upsilon_i}; (2^{15} \leq \rho_i < 2^{16}, \upsilon \in \mathbb{N}) \\ &V_i(t) := V_i(t) + (t_1 - t_0) \cdot \rho_i \cdot 2^{-\upsilon_i}; \end{aligned} \tag{4.22}$$

Code needs to be inserted into the application SW to collect and update specific performance information. Simulation accuracy can be severely impacted if time spent on such code is charged to application execution time, especially when detailed performance information is necessary. RTSP is implemented to allow application notify when to start/stop charging time usage so that the performance-collection time can be excluded.

Each *simcom* needs to provide the following two callback functions to the simulation scheduler: *preschedule()* and *postschedule()*. When a *simcom* is allowed to use an assigned Q, simulation scheduler calls its *preschedule()* in which the *simcom* can be simply resumed or an RTOS model can be executed. After using Q, its *postschedule()* is called. This is the place where *simcom* can update its state other than stopping itself.

The legacy DSP as *simhw* typically only supports 32 bit operation while it is often necessary to maintain 64-bit time information. For example, if the clock frequency is 1GHz, a 32-bit timer is only able to last 4.3s without rolling back. RTSP is implemented to support maintaining 64-bit time information on the 32-bit architecture.

Theorem 4.3.2 yields the delay bound for the worst case in the sense that $\forall$ c_i^k, 1) its length is always equal to the minimal length; 2) its start time is always delayed by D_i as shown in Equ. (4.10); and 3) its finish time is always delayed by $\triangle_i$. In reality such a worst case probably will not always happen. Therefore, the user should reduce Ks and S to speed up simulation while still maintaining the fidelity. RTSP is implemented to report warnings when the delay bound is violated because of the reduced K and S values.

4.7 Experiment

4.7.1 Audio Application

Two real DSP industry applications are selected for simulation. The first is an audio application [94] that produces delay effects and mixes them with the original samples. The application runs on the TMS320C6727 which is a floating point DSP from TI [20]. A high level block diagram of C6727 is shown in Fig. 4.10-(a).

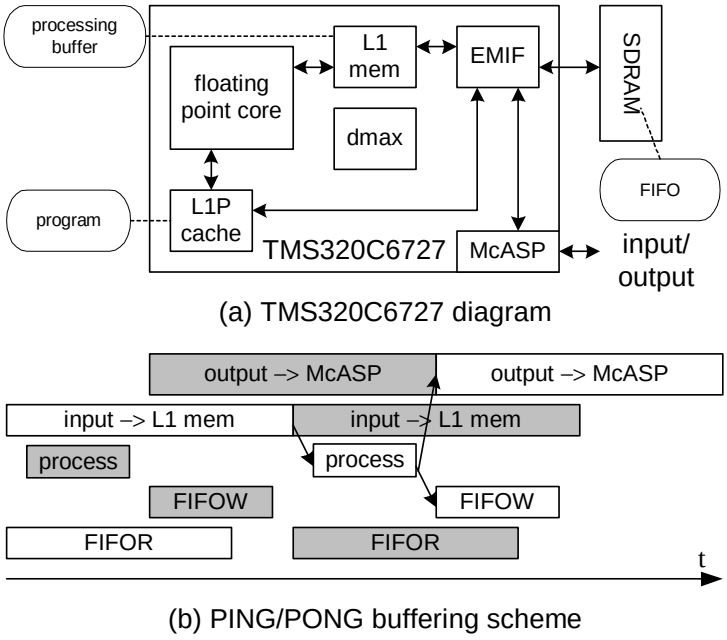

Figure 4.10: Audio Application

The application processes a block of input samples at a time. The program resides in the L1P cache. A processing buffer is allocated in L1 memory which holds all the data to be processed at this time. A circular buffer is allocated in the SDRAM which contains the delay samples to be processed and mixed with the current input block. It is updated in FIFO order in the sense that the latest delay samples always replace the oldest ones.

The input/output audio samples are transferred from/to McASP port to/from the L1 memory as general purpose (GP) transfer by *dmax* which is an application specific DMA engine. Other than GP transfer, *dmax* also manages FIFO read (FIFOR) and FIFO write (FIFOW) transfers. FIFOR brings all the delay samples to be mixed with the current input block from FIFO and packs them to the processing buffer. After the current block has been processed, FIFOW transfers the latest delay samples to FIFO. PING/PONG buffering scheme is deployed as shown in Fig. 4.10-(b). For example, when PING/PONG processing buffer is being processed, *dmax* transfers input and delay samples to PONG/PING processing buffer for the next round.

The simulation platform chosen is the TMS320C6455 EVM kit [18] with two C6455 DSPs, as shown in Fig. 4.11. Compared to C6727, C6455 has a fixed point DSP core, on-chip L2 memory, and SRIO for high speed inter-processor communication. $simcom_1$ is assigned to execute on DSP_1 to simulate the audio algorithm residing in L1P cache. $simcom_2$ is constructed on DSP_2 to simulate the FIFOR and FIFOW behavior of *dmax*. Some input samples are pre-stored in SDRAM of DSP_2 and the GP transfers handling input/output from/to McASP are not simulated. FIFO and processing buffer reside in SDRAM and L2 memory of DSP_2, respectively. After $simcom_2$ finishes a FIFOR, the data in the processing buffer are transferred to the L2 memory of DSP_1. Before $simcom_1$ processes the data, initially there will be a L1D cache miss which forces the data to be moved to L1D for processing. After processing is done, $simcom_1$ issues a cache writeback to L2 memory and then transfers the updated delay samples to $simcom_2$ via SRIO. FIFOW on $simcom_2$ transfers these samples to SDRAM to update FIFO.

Before simulation, the first step is to determine the speed ratio between each *simcom* and its corresponding *rhw*. For $simcom_1$, the ratio can be calculated as Equ. (4.23). C6455 and C6727 runs at 1GHz and 250MHz, respectively. For the audio algorithm, there are 26% floating point operations each of which takes 70

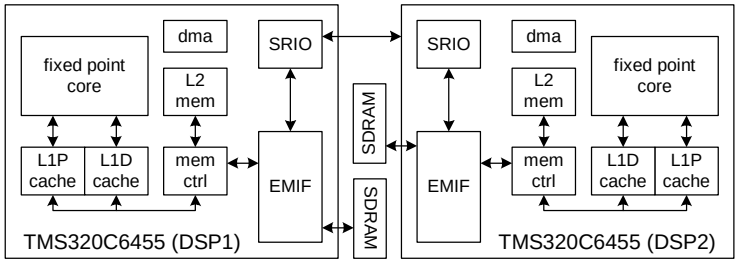

Figure 4.11: Simulation Hardware

cycles to be emulated on C6455 while on C6727 a native floating operation takes one cycle only.

$$\frac{speed(simcom_1)}{speed(C6727)} = \frac{\frac{1000}{74\%+70\cdot 26\%}}{250} = \frac{1}{4.74} \tag{4.23}$$

Finding the speed ratio for $simcom_2$ is more complicated. For FIFOW, the *dmax* is configured to transfer 4 samples a time. Reading each sample from L1 memory and writing it to SDRAM takes 1 and 4 DSP cycles, respectively. Managing the 4-sample transfer takes 6 cycles. These numbers can be obtained from the HW design specification. For $simcom_2$, reading a sample from L2 memory and writing it to SDRAM takes 6 and 44 DSP cycles, respectively. Managing the transfer takes about 420 cycles. These numbers are from benchmarks. Therefore, the ratio for FIFOW can be calculated by Equ. (4.24).

$$\frac{speed(simcom_{2,FIFOW})}{speed(dmax_{FIFOW})} = \frac{1000}{250} \cdot \frac{(1+4)+6/4}{(6+44)+420/4} = \frac{1}{5.96} \tag{4.24}$$

The calculation for FIFOR is similar but the difference is that dmax tranfers 8 samples a time, which makes the speed ratio 4.46 as shown in Equ. (4.25). To get a constant ratio between dmax and $simcom_2$, 5.96 is chosen. Appropriate amount of delay cycles are inserted after each FIFOR to make its ratio equal to 5.96.

$$\frac{speed(simcom_{2,FIFOR})}{speed(dmax_{FIFOR})} = \frac{1000}{250} \cdot \frac{(1+4)+6/8}{(6+44)+420/8} = \frac{1}{4.46} \tag{4.25}$$

Given Equ. (4.23) and (4.24), the minimal speed ratio between rhw execution and simulation is 5.96, that is, $R = 5.96$. α_1 and α_2 are computed by Equ. (4.26).

$$\alpha_1 = \frac{4.74}{5.96} = 0.80; \quad \alpha_2 = 1.0; \tag{4.26}$$

Before calculating K_1, K_2 and S, the minimal job length (P_1 and P_2) for each $simcom$ and the Q size need to be known. P_1 is profiled to be about $175\mu s$. Given the fact that FIFOR and FIFOW transfer 14 and 26 blocks respectively each time with block size being 32, P_2 can be estimated using Equ. (4.27).

$$P_2 = FIFOW + FIFOR = [(6 + 44) + \frac{420}{4}] \cdot (14 + 26) \cdot 32 = 198.4\mu s \tag{4.27}$$

The Q size of the operating system (DSP/BIOS) [15] on C6455 is 1ms. S, K_1 and K_2 are calculated to be 16.0, 20 and 16 respectively, using the search algorithm described in section 4.4. To speed up the simulation by reducing Q to 100μs, the timer on C6455 is re-programmed. S, K_1 and K_2 are reduced to 5.6, 7 and 5 respectively. They can be further reduced because of the pessimistic assumption of theorem 4.3.2. For example, if $Q = 100\mu s$, the simulation shows that using $K_1 = 2$, $K_2 = 1$ and $S = 1.6$ can still keep the delay in bound.

The simulation result is shown in Tab. 4.1. The total overhead including both first and second level scheduler is less than 1μs on average. The result for $simcom_1$ (process) is accurate for both cases. When $Q = 100\mu$s, the average transfer time for both FIFOR and FIFOW is still accurate but the minimal and maximal time show $20\% - 50\%$ errors. This is expected since the behavior model of $simcom_2$ cannot match the actual HW from timing perspective, i.e. the 420 cycles shown in Equ. (4.24) for transfer management is the average but not min/max number. When $Q = 1$ms, the average time for both FIFOR and FIFOW shown by simulation are both larger than the rhw execution number. The reason can be explained as following. For the rhw execution scenario, FIFOW of PING/PONG round overlaps

	$Q = 100\mu$s, $\gamma = 9.6$ $K_1 = 2$, $K_2 = 1$			$Q = 1$ms, $\gamma = 96$ $K_1 = 20$, $K_2 = 16$		
μs	min	max	avg	min	max	avg
FIFOR	62.0	123	92.2	104	124	117
FIFOW	21.0	70.7	33.6	21.5	65.1	51.0
process	34.0	41.3	37.7	32.9	38.9	36.4
schedule	0.466	1.746	0.855	0.472	1.728	0.853

	C6727 execution		
μs	FIFOR	FIFOW	process
min	90.1	30.5	36.8
max	106	46.0	46.8
avg	93.4	34.0	36.9

Table 4.1: Audio Simulation Result

only partially with FIFOR of PONG/PING round as shown in Fig.4.10-(b). During simulation when Q is large, FIFOR and FIFOW are always started at the same time when a new Q can be used by $simcom_2$, which makes FIFOW overlap completely with FIFOR. More overlap increases time for both transfers. A small Q size not only can speed up the simulation but also improve accuracy in this case.

4.7.2 Video Application

The second application is a MPEG2 decoder running on TMS320DM642 processor [21] which has almost the identical architecture comparing to C6455 except that DM642 is clocked at 600MHz. Each video frame is decoded MB by MB. The reference frame resides in SDRAM. When decoding the current MB, a reference MB needs to be transferred from SDRAM to on-chip L2 memory for fast processing. After being decoded, the current frame becomes the new reference frame and is moved to SDRAM.

The simulation platform is the same. The decoding algorithm is simulated on DSP_1 as $simcom_1$ and the DMA operation is simulated on DSP_2 as $simcom_2$. The

reference frame resides in SDRAM of DSP_2. To get a reference MB for $simcom_1$, the MB is moved from SDRAM to L2 memory of DSP_2 by $simcom_2$, and then transferred to L2 memory of DSP_1 via SRIO. Dataflow of updating the reference frame is in reverse.

Determining the speed ratio between $simcom_1$ and DM642 core is straightforward as shown in Equ. (4.28).

$$\frac{speed(simcom_1)}{speed(DM642)} = \frac{1000}{600} = \frac{1}{0.6} \tag{4.28}$$

The ratio between $simcom_2$ and DM642 DMA is shown in Equ. (4.29). Transferring a MB is a two dimensional transfer of 16×16 bytes. For DM642 DMA, the transfer management takes 20 cycles for each row. 4 bytes are accessed at a time to optimize the performance. Each access to L2 memory and SDRAM takes 2 and 33 cycles, respectively. For $simcom_2$, the management overhead for each MB is about 1120 cycles. Similarly, 4 bytes are transferred a time. Accessing time to L2 memory and SDRAM takes 6 and 44 cycles respectively.

$$\frac{speed(simcom_2)}{speed(DM642_{DMA})} = \frac{1000}{600} \cdot \frac{320 + 16 \cdot 2 \cdot (2 + 33)}{1120 + 16 \cdot 2 \cdot (6 + 44)} = \frac{1}{1.14} \tag{4.29}$$

Given Equ. (4.28) and (4.29), $R = 1.14$, $\alpha_1 = 0.53$ and $\alpha_2 = 1.0$.

The number of MBs to be decoded/transferred at a time is determined by the pre-encoded input video stream. The worst case is one, which makes $P_2 = 2.72\mu$s given by Equ. (4.29). P_1 can only be estimated and it is typically smaller than P_2. In this case, we assume it is 2μs.

For $Q = 100\mu$s, S, K_1 and K_2 are calculated to be 134, 253 and 134 respectively. For $Q = 1$ms, S, K_1 and K_2 are calculated to be 1322, 2503 and 1322 respectively. Since decoding/transferring one MB at a time rarely happens, simulation can be carried faster by reducing K_1, K_2 and S. The simulation result using the actual parameters are shown in Tab. 4.2.

	$Q = 100\mu$s, $\gamma = 30$ $K_1 = 50$, $K_2 = 25$			$Q = 1$ms, $\gamma = 180$ $K_1 = 300$, $K_2 = 150$		
	min	max	avg	min	max	avg
DMA(μs)	2.34	36.4	4.87	2.34	34.0	4.83
decode(ms)	11.7	25.2	22.0	11.5	24.7	21.8
schedule(μs)	0.394	5.338	0.933	0.412	4.788	0.922

	DM642 execution	
	DMA(μs)	decode (ms)
min	2.36	11.3
max	35.5	24.6
avg	4.80	21.5

Table 4.2: Video Simulation Result

The simulation accuracy of this application is higher than the audio application. The simulation error of decoder is $< 4\%$ due the architecture similarity between DM642 and C6455. The simulation error of DMA is $< 5\%$ even for the min/max case. It is because the DMA behavior is simple in this application and the SW behavior model is able to match well with the *rhw* from the timing perspective.

4.7.3 *simcom* Assignment Result

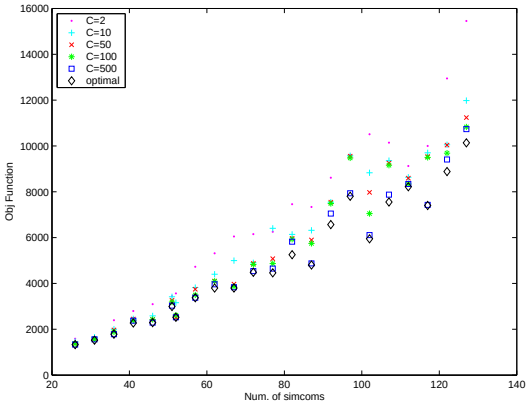

Figure 4.12: *simcom* Assignment Result: 4 *simhws*

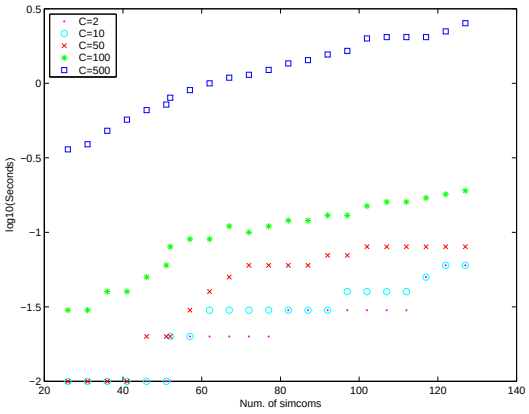

Figure 4.13: *simcom* Assignment Search Time: 4 *simhw*s

In this section, we present some experimental results on the *simcom* assignment algorithm. A random set of test cases are generated. $|\mathbb{H}|$ is in the range of $[20, 128]$. Constant C is set in the range of $[2, 500]$. $\forall\ simcom_i$, each of its job length p_i^k is a uniform distribution in $[5, 300]$ and $|\mathbb{C}_i|$ is set to 10. qc and sc are both set to 1. $\forall\ simhw_j$, the scheduler is set to be executed for 15 times. That is, $\lfloor \frac{t_{end}}{Q_j} \rfloor = 15$.

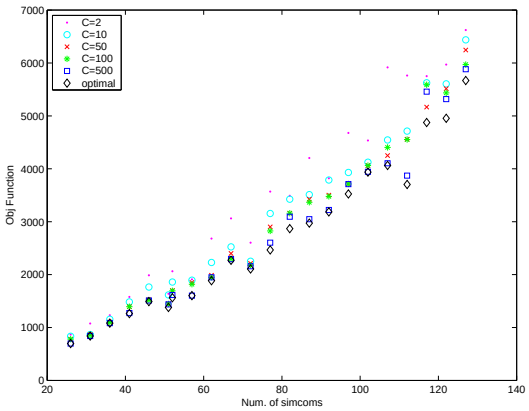

Figure 4.14: *simcom* Assignment Result: 8 *simhw*s

Fig. 4.12 and 4.14 plot the number of *simcom*s vs. the suboptimal solution found when the number of *simhw*s is set to 4 and 8, respectively. Fig. 4.13 and

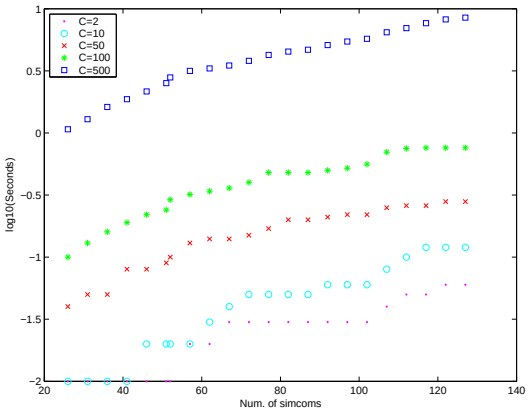

Figure 4.15: *simcom* Assignment Search Time: 8 *simhw*s

4.15 show the search time measured on the Toshiba Satellite laptop with a 1.33GHz Pentium III Cerelon CPU and 256 MB memory. The results demonstrate that the suboptimal solutions found by our algorithm are close to the optimal solutions in most cases when C is set to 500, while the search time is negligible.

4.8 Conclusion

To the best of our knowledge, RTSP is the first simulator that truly enables the application SW to be developed in parallel with HW. It offers the SW engineer the identical development environment as if the real HW (rhw) was available. Because legacy DSP cores are used in the simulation, the application SW can be directly compiled and run on RTSP without worrying about instruction set compatibility. The simulation components (*simcom*s) are scheduled to progress in unison, modulo bounded jitter. This eliminates unnecessary synchronization between the *simcom*s. A two-level scheduler is used with low overhead. The first level scheduler is a real-time periodic scheduler that assigns quantums (Qs) to the *simcom*(s). The second level scheduler bounds the completion delay for each job being simulated to achieve timing accuracy. RTSP is proved to perform simulations faithfully and also is showed

to be effective applying it to real industry applications. For a rhw whose timing behavior can be accurately modeled by the SW behavior model, the simulation error is shown to be $< 5\%$. For very complicated rhw whose timing cannot be accurately captured by the behavior model, the simulation accuracy was shown to be excellent for the average case. The simulation speed is quite fast. For the selected audio and video applications, simulation is only $10X$ and $30X$ slower than rhw execution. The RTSP platform is practically zero-cost since legacy EVM boards can be reused for the purpose of simulation.

Chapter 5

Conclusion

A trend in the consumer electronics market is the demand for new applications that have a lot of similarities to older applications but the new ones impose more challenging and special-purpose performance requirements. In the DSP industry, this clearly reflects a transition from the design regime of general DSP to the application-specific DSP. From the design perspective, this means that the DSP core remains unchanged but more and more HW accelerators, DMAs and bus architectures need to be integrated into the chip. A key in effecting this transition is the engineering capability to make sure the design specification "matches" the application before detailed design starts. Therefore, application SW needs to be developed in parallel with HW to verify the design specification at the system level. Enabling development and simulation of SW before the actual HW is available also reduces the time-to-market period which is another important benefit.

HW/SW co-simulation for design specification refinement imposes many challenging requirements to the simulation platform. HW and SW needs to be integrated to carry out the simulation at the system level. Simulation result needs to be accurate. Simulation speed should allow fast DSE and ease debugging complex application SW. HW and SW problems should isolated cleanly since in practice HW

and SW engineers often do not have enough expertise in one another's domains. The simulator should be cost-effective. These are often conflicting requirements and cannot be met by a single all-purpose simulator Instead, this dissertation proposes three simulators for different usages.

An RTOS tool is presented to model the RTOS with application SW to help generate an initial specification. It is motivated by the fact that more and more tasks are chosen to be implemented as SW on a single DSP managed by an RTOS. Selecting the "right" RTOS before the SW is developed is very important. The proposed tool is implemented based on SystemC and is configurable to support modeling and timed simulation of most popular embedded RTOSes. Timing fidelity is achieved by using delay annotations. The OS timing information is derived from published benchmark data. Application timing information can be profiled or estimated from similar legacy applications. The optimized conservative approach is taken to synchronize *simcom*s. Compared to other research work, an important contribution of this tool is an online algorithm for predicting the timestamp of the next event based on the realistic assumption that multiple tasks execute concurrently on a processor, managed by a static or dynamic priority driven scheduler. The simulation speed is more than 3 orders of magnitude faster than commercial the ISS with comparable accuracy.

The second tool is a system dataflow simulator (SDFS) to help HW engineers refine the HW specifications. It models the application by a parameter-driven conditional dataflow graph (CDFG) at the transaction level and the HW by a configurable HW graph at the cycle-accurate level. SDFS takes the application CDFG and HW graph as the input and carries out the simulation to catch the detailed HW activities. It only requires the HW engineers to understand the application SW at the CDFG level. To carry out the system simulation at such a low level, many commercial simulators need to couple an ISS for the application SW with an RTL simulator for the *simcom*s to model the rhws which is typically 6 orders of magni-

tude slower than the rhw speed. Our simulation error of SDFS is within 5% in most cases and the worst case error is within 13% which is comparable to the ISS+RTL approach. But our simulation speed is only 4 orders of magnitude slower than the rhw execution. Compared with other similar research work that also models the system at the CDFG level, SDFS can achieve higher simulation accuracy because of the following advantages: 1) it does not need a fixed application trace as input and thus is flexible enough to cover many simulation scenarios; 2) it does not assume a fixed cost for each block and thus is able to estimate the system performance under actual execution conditions; and 3) it is able to model pipelined architecture common in modern DSPs. SDFS is cost-effective since it is implemented in the SystemC language and can be executed on almost any PCs and workstations.

The third tool is a real-time simulation platform (RTSP) implemented on legacy DSPs. To the best of our knowledge, this is the first simulator that truly enables the application SW to be developed in parallel with HW by offering the same SW development environment as if the rhw is available. To simulate the behavior of a rhw module, a corresponding $simcom$ is constructed running on a legacy DSP. The success of this simulation strategy hinges on a novel way to apply the concept of Real-Time Virtual Machines to simulation. Each legacy DSP employs a two-level scheduler to enforce the policy that each $simcom$ carries out the simulation at a proportional speed $(1/\gamma)$ of the rhw, so that any job that would finish at time t on the rhw will finish no later than $\gamma \cdot t + \triangle$ where $\triangle$ is a constant bound. Such a feature eliminates expensive synchronization between the $simcom$s. RTSP is proved to perform simulations faithfully and also is shown to be effective by application to real industry applications. For a rhw whose timing behavior can be accurately modeled by the SW behavior model, the simulation error is shown to be $< 5\%$. For very complicated rhw whose timing cannot be accurately captured by the behavior model, the simulation accuracy was shown to be excellent for the average case. The simulation speed is quite fast. For the selected audio and video applications,

simulation is only $10X$ and $30X$ slower than rhw execution. The RTSP platform is practically zero-cost since legacy EVM boards can be reused for the purpose of simulation.

RTSP and SDFS can be used to complement each other. RTSP carries the simulation at a higher level than SDFS and usually cannot capture activities on buses at every cycle. The information collected from SDFS determines appropriate rate settings to the *simcom*s to compensate for the bus activities. RTSP allows SW engineers optimize the algorithm and suggest improvements to the HW architecture. The suggested changes can be fed to SDFS to refine the design specification.

Appendix A

Acronyms

ASIC: application specific integrated circuit

CAD: computer-aided design

CE: condition edge

CFG: conditional-flow graph

DE: data edge

CFSM: co-design finite state machine

DFG: dataflow graph

DFT: design-for-test

DSE: design space exploration

EDA: electronic design automation

EVM: evaluation version module

FIFO: first-in-first-out

FIFOR: FIFO read

FIFOW: FIFO write

FPGA: field programmable gate array

FSM: finite state machine

GOB: group of blocks

GP: general purpose

GPP: general purpose processor

HDL: hardware description language

HW: hardware

ILP: instruction level parallelizing

IOM: I/O manager

IPC: inter-process communication

ISR: interrupt service routine

IST: interrupt service thread

MB: macroblock

MP-SoC: multiprocessor SoC

NoC: networks on chips

OS: operating system

RISC: reduced instruction set computer

RTL: register transfer level

RTOS: real-time operating system

RTSP: real-time simulation platform

SDFS: system dataflow simulator

SMP: symmetric multi-processor

SoC: system on chip

SRIO: serial rapid I/O

SW: software

TI: Texas Instruments

TLM: transaction level model

VLIW: very long instruction word

Appendix B

Notations

- $\mathbb{H}$ is the set of real HW modules to be designed. The number of elements in $\mathbb{H}$ is denoted as $|\mathbb{H}|$ and rhw_i is i^{th} element in $\mathbb{H}$. (notation 1.5.1)
- $\forall\ rhw_i \in \mathbb{H}$, $simcom_i$ is a simulation component modeling its behavior and timing characteristics in the specified abstract level. (notation 1.5.2)
- $\mathbb{SH}$ is the set of simulation HW on which simulation is being carried. The number of elements in $\mathbb{SH}$ is denoted as $|\mathbb{SH}|$ and $simhw_i$ is i^{th} element in $\mathbb{SH}$. $\forall\ simcom_j$, it is simulated on a $simhw$ in $\mathbb{SH}$. If $simcom_j$ is simulated on $simhw_i$, it is denoted as $simcom_j \in simhw_i$. (notation 1.5.3)
- t_{end} is the time that simulation ends. (notation 1.5.4)
- $func$ is a function block realizing certain functionality. It will be implemented on a rhw and simulated on the corresponding $simcom$. (notation 1.5.5)
- T is a task which consists of one or multiple $func$s and has its own execution context. (notation 1.5.6)
- $\forall\ func$, its execution instance is called a job. $\forall\ simcom_i$, $\mathbb{C}_i$ is the set of job(s) to be simulated on it during $[0, t_{end}]$. The number of job(s) is denoted

as $|\mathbb{C}_i|$ and the k^{th} job is denoted as c_i^k. (notation 1.5.7)

- $\mathbb{R}$ and $\mathbb{N}$ represent the real number set and the non-negative integer set, respectively. (notation 1.5.8)

B.1 Notations in RTSP

- $\gamma = R \cdot S$ is the ratio between the speed of $\mathbb{H}$ and simulation, where R is the minimal achievable ratio determined by $\mathbb{SH}$ without considering bounding delay, and S is the slowdown factor to bound delay. (notation 4.3.1)

- α_i and K_i are the scheduling parameters of $simcom_i$. It progresses at rate $\frac{\alpha_i}{S}$ but receives supply from its $simhw$ at rate $\frac{\alpha_i \cdot K_i}{S}$. $K_i \in \mathbb{N}$ and $K_i \geq 1$. (notation 4.3.2)

- Q is the scheduling quantum determined by a $simhw$. Q_i denotes the quantum for $simcom_i$. (notation 4.3.3)

- L_i measures the difference between the actual and normal supply of $simcom_i$. (notation 4.3.4)

- $V_i(t)$ is $simcom_i$'s virtual time. (notation 4.3.5)

- $c_i^k = \{p_i^k, \mathcal{E}_i^k, t\varepsilon_i^k, (ts_i^k, Vs_i^k), (te_i^k, Ve_i^k)\}$ (notation 4.3.6)

  c_i^k is the k^{th} job on $simcom_i$.

  - ts_i^k (te_i^k): actual start (finishing) time of c_i^k.
  - Vs_i^k (Ve_i^k): virtual start (finishing) time of c_i^k.
  - p_i^k: length of c_i^k.
  - $\mathcal{E}_i^k$: the set of event(s) $simcom_i$ needs to receive before c_i^k starts. $\varepsilon_j^l \in \mathcal{E}_i^k$ means the the event is sent by $simcom_j$ at the end of c_j^l.

- $t\varepsilon_i^k$: time that the last event in $\mathcal{E}_i^k$ is received during simulation.

- Θ_i is the set of $simcom$(s) that will send event(s) to $simcom_i$. (notation 4.3.7)
- $\triangle_i = \frac{2 \cdot L_i \cdot Q_i \cdot S}{K_i \cdot \alpha_i}$ is the delay bound for $simcom_i$. (notation 4.3.8)
- $state_i(t)$ is the state of $simcom_i$ at time t. It can either be $exec_i^k$ meaning that $simcom_i$ is simulating c_i^k, or $idle_i^k$ meaning that $simcom_i$ is waiting for c_i^k to start. (notation 4.3.9)
- $t \rightarrow i$ / $\overline{t \rightarrow i}$ represents that a Q with interval $[t, t + Q_i]$ is / is not assigned to $simcom_i$. (notation 4.3.10)
- $t \rightrightarrows i$ / $\overline{t \rightrightarrows i}$ means that $simcom_i$ is / is not allowed to use the Q assigned at t. (notation 4.3.11)
- $u_i[t_0, t_1]$ indicates that $simcom_i$ actually used a Q from in $[t_0, t_1]$. (notation 4.3.12)
- A partial solution is called a $\langle n, m, k \rangle$ *partial solution* if n *simcom*s have been assigned to m *simhw*s among which k *simhw*s are not assigned any *simcom*. We call those k *simhw*s *idle simhw*s. The other $(m - k)$ *simhw*s are called *busy simhw*s. (notation 4.5.1)
- The most *promising* $\langle n, m, k \rangle$ *partial solution* among all $\langle n, m, i \rangle$ partial solutions is defined as the one which achieves the minimum of Equ. (4.15), only considering the n *simcom*s assigned and the m *simhw*s. (notation 4.5.2)
- $\mathbb{B}_{\langle n,m,k \rangle}$ is a set containing all the promising $\langle n, m, k \rangle$ partial solutions being kept. Its size is denoted as $|\mathbb{B}_{\langle n,m,k \rangle}|$. (notation 4.5.3)
- $\mathbb{A}_{\langle n,m,k \rangle}$ is a set which containing all the $\langle n, m, k \rangle$ partial solutions derived from the partial solutions kept in the previous search step. Its size is denoted as $|\mathbb{A}_{\langle n,m,k \rangle}|$. (notation 4.5.4)

Bibliography

[1] "Altera," http://www.altera.com.

[2] "Broadway compiler project," http://www.cs.utexas.edu/users/less/broadway.html.

[3] "CarbonKernel," http://www.carbonkernel.org.

[4] "Cocentric systemc compiler," http://www.synopsys.com.

[5] "A framework for hardware-software co-design of embedded systems," http://embedded.eecs.berkeley.edu/Research/hsc/abstract.html.

[6] "Functinal specification for systemc 2.0," http://www.systemc.org.

[7] "H.263 decoder: TMS320C6000 implementation," http://www.ti.com.

[8] "International technology roadmap for semiconductors," http://www.public.itrs.net.

[9] "Mentor graphics," http://www.mentor.com.

[10] "Open core protocol specification 2.1," http://www.ocpip.org.

[11] "quickturn," http://www.cadence.com/quickturn/.

[12] "Serial RapidIO," http://www.rapidio.org.

[13] "Tensilica Incorporated," http://www.tensilica.com.

[14] "AMBA specification (rev 2.0)," ARM Ltd. 1999.

[15] "DSP/BIOS kernel technical overview," http://www.ti.com.

[16] "DSP/BIOS timing benchmarks for code composer studio 2.2," http:// www.ti.com.

[17] "IEEE 1500 standard," http://www.grouper.ieee.org/groups/1500.

[18] "TMS320C6455 fixed-point digital signal processor," http://www.ti.com.

[19] "TMS320C64x image/video processing library programmer's reference," http://www.ti.com.

[20] "TMS320C6727, TMS320C6726, TMS320C6722 floating-point digital signal processors," http://www.ti.com.

[21] "TMS320DM642 video/imaging fixed-point digital signal processor," http://www.ti.com.

[22] "VHDL 93 references," http://www.vhdl-online.de/ref93/.

[23] "Virtual component codesign," Cadence Design Systems Inc.

[24] "Virtual socket interface alliance," http://www.vsi.org.

[25] "VxWorks 5.4," http://www.wrs.com/products/html/vxwks54.html.

[26] "Xilinx," http://www.xilinx.com.

[27] "Open verification library assertion monitor reference manual," AcceUm, 2002.

[28] "Superlog design assertion subset," Co-design Automation Ins., Apri. 2002.

[29] H.M. AbdElSalam, S. Kobayashi, K. Sakanushi, Y. Takeuchi, and M. Imai, "Towards a higher level of abstraction in hardware/software co-simulation," in *24th International Conference on Distributed Computing Systems Workshops*, pp. 824–830, 2004.

[30] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*, MA: Addison-Wesley, 1988.

[31] H. Akaboshi, *A Study on Design Support for Computer Architecture Design*, Ph.D. thesis, Department of lnformation Systems, Kyushu University, 1996.

[32] Guido Arnout, "C for system level design," in *Design, Automation and Test in Europe Conference and Exhibition*, pp. 384–386, 1999.

[33] J. Axelsson, "Towards system-level analysis and synthesis of distributed real-time systems," in *5th International Conference on Information Systems Analysis and Synthesis*, pp. 40–46, 1999.

[34] A. Baghdadi, N.E. Zergainoh, W.O. Cesario, and A. A. Jerraya, "Combining a performance estimation methodology with a hardware/software codesign flow supporting multiprocessor systems," *IEEE Trans. Software Engineering*, vol. 28, no. 9, pp. 822–831, 2002.

[35] Jwahar R. Bammi, Wido Kruijtzer, and Luciano Lavagno, "Software performance estimation strategies in a system-level design tool," in *8th International Workshop on Hardware/Software Codesign*, pp. 82–86, May 2000.

[36] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer, "The MIMOLA language version 4.1," University of Dortmund, 1994.

[37] Matthias Bauer, Wolfgang Ecker, Michael Gasteir, and Manfred Glesner,

"Evaluation of sequential VHDL and C for system description and specification," in VIUF, 1996.

[38] David Becker, Raj K. Singh, and Stephen G. Tell, "An engineering environment for hardware/software co-simulation," in *29th Design Automation Conference*, June 1992.

[39] Ilan Beer, Shoham Ben-David, Cindy Eisne, and Amer Landvn, "Rule base: An industry-oriented formal verification tool," in *Design Automation Conference*, pp. 655–660, June 1996.

[40] A. Bender, "Design of an optimal loosely coupled heterogeneous multiprocessor system," in *Proc. EDTC*, pp. 275–281, 1996.

[41] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "Legacy systemc co-simulation of multi-processor systems-on-chip," in *IEEE International Conference on Computer Design*, pp. 494–499, 2002.

[42] B. Bentley, "High level validation of next generation microprocessors," in *IEEE International Workshop on High-Level Design Validation and Test*, pp. 31–35, 2002.

[43] S. S. Bhattacharyya, S. Sriram, and E.A. Lee, "Optimizing synchronization in multiprocessor DSP systems," *IEEE Transaction on Signal Processing*, vol. 45, no. 6, pp. 1605–1618, Jun. 1997.

[44] Lubomir F. Bic and Alan C. Shaw, *Operating Systems Principles*, Prentice Hall, 2002.

[45] N. Binh, M. Imai, A. Shiomi, and N. Hickichi, "A HW/SW partitioning algorithm for designing pipelined asip's with least gate counts," in *Proc. DAC*, pp. 527–532, 1996.

[46] G. Bosman, *A Survey of Co-Design Ideas and Methodologies*, Ph.D. thesis, Vrije University, Amsterdam, Netherland, 2002.

[47] A. Bouchhima, S. Yoo, and A. Jerraya, "Fast and accurate timed execution of high level embedded software using HW/SW interface simulation model," in *Asia and South Pacific Design Automation Conference*, pp. 469–474, Jan. 2004.

[48] F. Boussinot and R. De Simone, "The ESTEREL language," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, Sept. 1991.

[49] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, and D. Sciuto, "Affinity-driven system design exploration for heterogeneous multiprocessor soc," *Computers*, vol. 55, no. 5, pp. 508–519, May 2006.

[50] L. Carro, M. Kreutz, F. R. Wagner, and M. Oyamada, "system synthesis for multiprocessor embedded applications," in *DATEC*, pp. 687–702, 2000.

[51] Noureddine Chabini, Imed Eddine Bennour, El Mostapha Aboulhamid, and Yvon Savaria, "A static method for system performance estimation," in *10th International Conference on Microelectronics*, pp. 111–114, Dec. 1998.

[52] A. Chakraborty and M. Greenstreet, "Efficient self-timed interfaces for crossing clock domains," in *IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 78–88, May 2003.

[53] Nelson Yen-Chung Chang, Kun-Bin Lee, and Chien-Wei Jen, "Trace-path analysis and performance estimation for multimedia application embedded system," in *International Symposium on Circuits and Systems*, 2004.

[54] W.-T. Chang, S. Ha, and E. A. Lee, "Heterogeneous simulation - mixing discrete-event models with dataflow," *VLSI Signal Processing*, vol. 15, pp. 127–144, 1997.

[55] Karam S. Chatha and Ranga Vemuri, "Hardware-software partitioning and pipelined scheduling of transformative applications," *IEEE Trans. on VLSI*, pp. 193–208, 2002.

[56] T. Chelcea and S. Nowick, "Robust interfaces for mixed-timing systems," *IEEE Trans. on VLSI Systems*, vol. 12, no. 8, pp. 857–873, Aug. 2004.

[57] M. Chiodo, D. Engels, P. Giusto, A. Jurecska, H. Hsieh, L. Lavagno, K. Suzuki, and A. Sangiovanni, "A case study in computer-aided co-design of embedded controllers," *Design Automation for Embedded Systems*, vol. 1, no. 2, pp. 51–67, Jan. 1996.

[58] Moo-Kyoung Chung and Chong-Min Kyung, "Enhancing performance of HW/SW cosimulation and coemulation by reducing communication overhead," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 125–136, Feb. 2006.

[59] Moo-Kyoung Chung, Sangjun Yang, Sang-Hoon Lee, and Chong-Min Kyung, "System-level HW/SW co-simulation framework for multiprocessor and multithread SoC," in *IEEE International Symposium on VLSI Design, Automation and Test*, pp. 177–180, Apri. 2005.

[60] E. Clarke, D. Long, and K. McMillan, "Compositional model checking," in *4th Annual Symposium on Logic in Computer Science*, pp. 353–362, 1989.

[61] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, Cambridge, MA: MIT Press, 2000.

[62] J. Cong and Y. Ding, "Combinational logic synthesis for LUT based field programmable gate arrays," *ACM Trans. Design Automation for Electronic Systems*, vol. 1, no. 2, pp. 145–204, Apri 1996.

[63] B. P. Dave and N. K. Jha, "COHRA: Hardware/software co-synthesis of hierarchical heterogenous distributed embedded systems," *IEEE Trans. Computer-Aided Design*, vol. 17, Oct. 1998.

[64] J. A. Debardelaben and V. K. Madisetti, "Hardware/software codesign for signal processing systems - a survey and new results," in *the 29th Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp. 1316–1320, Nov. 1995.

[65] G. DeMicheli, *Synthesis and Optimization of Digital Circuits*, New York: McGraw-Hill, 1994.

[66] D. Desmet, D. Verkest, and H. De Man, "Operating system based software generation for system-on-chip," in *Design Automation Conf.*, pp. 396–401, Jun. 2000.

[67] R. P. Dick and N. K. Jha, "Mogac: A multiobjective genetic algorithm for hardware/software cosynthesis of distributed embedded systems," *IEEE Trans. Computer-Aided Design*, vol. 17, Oct. 1998.

[68] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, 2002.

[69] P. Eles, "VHDL system-level specification and partitioning in a hardware/software co-synthesis environment," in *International Workshop on Hardware/Software Codesign*, pp. 22–24, Sept. 1994.

[70] P. Eles, Z. Peng, K. Kuchinski, and A. Doboli, "System level hardware/software partitioning based on simulated annealing and tabu search," *Design Automation for Embedded Systems*, vol. 2, no. 2, pp. 5–32, 1997.

[71] R. Ernst, J. Henkel, and T. Benner, "Hardware/software cosynthesis for microcontrollers," *IEEE Design and Test of Computers*, pp. 64–75, Dec. 1993.

[72] R. Ernst, J. Henkel, Th. Benner, W. Ye, U. Holtmann, D. Herrmann, and M. Trawny, "The COSYMA environment for hardware-software cosynthesis of small embedded systems," *Microprocessors and Microsystems*, May 1996.

[73] L. Formaggio, F. Fummi, and G. Pravadelli, "A timing-accurate HW/SW cosimulation of an ISS with SystemC," in *International Conference on Hardware/Software Codesign and System Synthesis*, pp. 152 – 157, 2004.

[74] Richard M. Fujimoto, "Parallel discrete event simulation," in *Winter Simulation Conference*, pp. 19–28, 1989.

[75] R. M. Fujimoto, "Time warp on a shared memory multiprocessor," in *International Conference on Parallel Processing*, 1989.

[76] D. Gajski, R. Domer, and J. Zhu, "IP-centric methodology and design with the SpecC language," 1998.

[77] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, S*pec*C*: Specification Language and Methodology*, Kluwer Academic Publishers, 2000.

[78] C. Gebotys and M. Elmasry, *Optimal VLSI Architectural Synthesis*, Amsterdam: Kluwer, 1992.

[79] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski, "RTOS modeling for system level design," in *Design, Automation and Test in Europe Conference and Exhibition*, pp. 130–135, 2003.

[80] G. Goossens, P. G. Paulin, J. Van Praet, D. Lanneer, W. Guerts, A. Kifli, and C. Liem, "Embedded software in real-time signal processing systems: Design technologies," *Proc. IEEE*, pp. 436–454, 2001.

[81] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan, *System Design with SystemC*, Kluwer Academic Publishers, 2002.

[82] Lisa Guerra, Joachim Fitzner, Dipankar Talukdar, Chris Schlager, Bassam Tabbara, and Vojin Zivojnovic, "Cycle and phase accurate DSP modeling and integration for HW/SW co-verification," in *Design Automation Conference*, pp. 964–969, 1999.

[83] Pallav Gupta, "Hardware-software codesign," *IEEE Potentials*, vol. 20, no. 5, pp. 31–32, Dec. 2001.

[84] R. Gupta, C. Coelho, and G. DeMicheli, "Program implementation schemes for hardware-software systems," *IEEE Computer*, pp. 48–55, Jan. 1994.

[85] R. K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 29–41, Sept. 1993.

[86] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability," in *34th Design Automation Conference*, pp. 299–302, 1997.

[87] G. Hadjiyiannis, P. Russo, and S. Devadas, "A methodology for accurate performance evaluation in architecture exploration," in *36th Design Automation Conference*, pp. 927–932, 1999.

[88] L. Hafer and A. Parker, "Automated synthesis of digital hardware," *IEEE Trans. Computers*, vol. C-31, no. 2, Feb. 1982.

[89] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in *Design Automation and Test in Europe*, 1999.

[90] A. Halambi, P. Grun, H. Tomiyama, N. Dutt, and A. Nicolau, "Automatic software toolkit generation for embedded systems-on-chip," in *6th International Conference on VLSI and CAD*, pp. 107–116, Oct. 1999.

[91] S. Hanono and S. Devadas, "Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator," in *35th Design Automation Conference*, pp. 510–515, 1998.

[92] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A working environment for the development of complex reactive systems," *IEEE Transactions on Sofware Engineering*, vol. 16, no. 4, pp. 403–414, April 1990.

[93] M. R. Hartoog, J. A. Rowson, P. D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, and N. Khullar, "Generation of software tools from processor descriptions for hardware/software codesign," in *34th Design Automation Conference*, pp. 303–306, 1997.

[94] Zhengting He, "How to create delay-based audio effects on a TMS320C6727 DSP," http://www.ti.com.

[95] Zhengting He, Aloysius K, and Mok, "Fast cosimulation of transformative systems with OS support on SMP computer," in *Hardware/Software Codesign and System Synthesis*, pp. 164–169, 2004.

[96] Zhengting He, Al. Mok, and C. Peng, "Timed RTOS modeling for embedded system design," in *11th IEEE Real Time Application and System Symposium*, pp. 448–457, Mar. 2005.

[97] Zhengting He and Aloysius K. Mok, "A real time simulation platform for hardware/software codesign," in *submitted to EMSOFT*, 2007.

[98] Zhengting He, C. Peng, and Al. Mok, "A performance estimation tool for video applications," in *12th Real-Time and Embedded Technology and Applications Symposium*, pp. 267–276, Apri. 2006.

[99] Dan Henriksson, Ola Redell, Jad El-Khoury, Martin Törngren, and Karl-Erik Årzén, "Tools for real-time control systems co-design - a survey," http://www.cse.unt.edu/ sweany/CoDesign/Hendricksson05.pdf.

[100] K. Hines and G. Borriello, "Dynamic communication models in embedded system co-simulation," in *Design Automation Conf.*, pp. 395–400, Jun. 1997.

[101] K. Hines and G. Borriello, "A geographically distributed framework for embedded system design and validation," in *Design Automation Conf.*, pp. 140–145, Jun. 1998.

[102] A. Hoffmann, T. Kogel, and H. Meyr, "A framework for fast hardware-software co-simulation," in *Design, Automation and Test in Europe*, pp. 760–764, Mar. 2001.

[103] S.-Y. Huang and K.-T. Cheng, *Formal Equivalence Checking and Design Debugging*, Boston, MA: Kluwer, 1998.

[104] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng, "overview of the ptolemy project," http://ptolemy.eecs.berkeley.edu/publications/papers/03/overview/overview03.pdf.

[105] A. Inoue, H. Tomiyama, H. Okuma, H. Kanbara, and H. Yasuura, "Language and compiler for optimizing datapath widths of embedded systems," IEICE *Trans. Fundamentals*, vol. E81-A, no. 12, pp. 2595–2604, Dec. 1998.

[106] A. Jantasch, P. Ellervee, J. Oberg, A. Hemani, and H. Tenhunen, "Hardware/software partitioning and minimizing memory interface traffic," in *EURODAC*, pp. 226–231, 1994.

[107] J. Jeon and K. Choi, "Loop pipelining in hardware/software partitioning," in *ASPDAC*, 1998.

[108] Jinyong Jung, Sungjoo Yoo, and Kiyoung Choi, "Performance improvement of multi-processor systems cosimulation based on SW analysis," in *Design, Automation and Test in Europe*, pp. 749–753, Mar. 2001.

[109] A. Kalavade and E. A. Lee, "A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem," in *3rd International Workshop on Hardware/Software Codesign*, pp. 42–48, Sept. 1994.

[110] M. Keating and P. Bricaud, *Reuse Methodology Manual: For System-on-a-Chip Designs*, 3rd edition, Boston, MA: Kluwer, 2002.

[111] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau, "V-SAT: A visual specification and analysis for system-on-chip exploration," in EUROMICRO, 1999.

[112] Dohyung Kim, Chan-EunRhee, Youngmin Yi, Sungchan Kim, Hyunguk Jung, and Soonhoi Ha, "Virtual synchronization for fast distributed cosimulation of dataflow task graphs," in *15th International Symposium on System Synthesis*, pp. 174–179, Oct. 2002.

[113] Dohyung Kim, Chan-Eun Rhee, and Soonhoi Ha, "Combined data-driven and event-driven scheduling technique for fast distributed cosimulation," *IEEE Transaction on VLSI*, vol. 10, no. 5, pp. 672–679, Oct. 2002.

[114] Dohyung Kim, Youngmin Yi, and Soonhoi Ha, "Trace-driven HW/SW cosimulation using virtual synchronization technique," in *42nd Design Automation Conference*, pp. 345–348, June 2005.

[115] J. Kim and Y. Kim, "Simulating multimedia systems with MVPSIM," *IEEE Transaction on Design and Test of Computers*, vol. 12, no. 4, pp. 18–27, 1995.

[116] Sungchan Kim, Chaeseok Im, and Soonhoi Ha, "Schedule-aware performance estimation of communication architecture for efficient design space exploration," in *Hardware/Software Codesign and System Synthesis*, pp. 195–200, Oct. 2003.

[117] Christian Kreiner, Christian Steger, Egon Teiniker, and Reinhold Weiss, "A HW/SW codesign framework based on distributed DSP virtual machines," in *Euromicro Symposium on Digital Systems Design*, pp. 212–219, Sept. 2001.

[118] D. Ku and G. De Micheli, "Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits," *IEEE Trans. CAD/ICAS*, pp. 696–718, June 1992.

[119] E. D. Lagnese and D. Thomas, "Architectural partitioning for system level descriptions," in *Proc. DAC*, pp. 62–67, 1989.

[120] Marcello Lajolo, Mihai Lazarescu, and Alberto Sangiovanni-Vincentelli, "A compilation-based software estimation scheme for hardware/software co-simulation," in *7th International Workshop on Hardware/Software Codesign*, pp. 85–89, May 1999.

[121] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens, *CHESS: Retargetable Code Generation for Embedded DSP Processors, In Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995.

[122] E. A. Lee and A. Sangiovanni-Vicentelli, "Comparing models of computation," in *ICCAD*, pp. 234–241, 1996.

[123] R. Leupers and P. Marwedel, "Retargetable code generation based on structural processor descriptions," *Design Automation for Embedded Systems, Kluwer Academic Publishers*, vol. 3, no. 1, pp. 1–36, Jan. 1998.

[124] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of ACM*, vol. 20, no. 1, 1973.

[125] Jie Liu, Marcello Lajolo, and A. Sangiovanni-Vincentelli, "Software timing analysis using HW/SW cosimulation and instruction set simulator," in *6th International Workshop on Hardware/Software Codesign*, pp. 65–69, Mar. 1998.

[126] X. Liu, J. Liu, J. Eker, and E. A. Lee, "Heterogeneous modeling and design of control systems," *Software-Enabled Control: Information Technology for Dynamical Systems*, 2002.

[127] R. Mateos, J. L. Lazaro, and F. Espinosa, "Hardware/software co-simulation environment for CSoC with soft processors," in *IEEE International Conference on Field-Programmable Technology*, pp. 445–448, 2004.

[128] S. A. Maxwell, *Linux Core Kernel 2nd Edition*, Coriolis Technology Press, 2001.

[129] D. Messerschmitt, "Synchronization in digital systems design," *IEEE Trans. on Selected Areas in Communications*, vol. 8, no. 8, pp. 1404–1419, Oct. 1990.

[130] G. De Michell and R.K. Gupta, "Hardware/software co-design," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 349–365, 1997.

[131] R. Le Moigne, O. Pasquier, and J.-P. Calvez, "A generic RTOS model for real-time systems simulation with SystemC," in *Design, Automation and Test in Europe Conference and Exhibition*, vol. 3, pp. 82–87, 2004.

[132] A. Mok and A. X. Feng, "Real-time virtual resource: A timely abstraction for embedded systems," in *EMSOFT*, pp. 182–196, 2002.

[133] Al. Mok, X. Feng, and Zhengting He, "Implementation of real-time virtual CPU partition on Linux," in *7th Real-Time Linux Workshop*, 2005.

[134] J. Mutterbach, T. Villiger, and W. Fichtner, "Practical design of globally-asynchronous, locally-synchronous systems," in *International symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 52–59, 2000.

[135] G. Nicolescu, Sungjoo Yoo, A. Bouchhima, and A. A. Jerraya, "Validation in a component-based design flow for multicore SoCs," in *International Symposium on System Synthesis*, 2002.

[136] R. Niemann and P. Marwedel, "Hardware/software partitioning using integer programming," in *EDTC*, pp. 473–479, 1996.

[137] J. Noguera, L. Baldez, N. Simon, and L. Abello, "Software-friendly HW/SW co-simulation: An industrial case study," in *Design, Automation and Test in Europe*, vol. 2, pp. 1–6, Mar. 2006.

[138] M. G. Norman and P. Thanisch, "Models of machines and computation for mapping in multicomputers," *ACM Computing Surveys*, vol. 25, no. 3, pp. 263–302, 1993.

[139] C. Passerone, L. Lavagno, C. Sansoe, M. Chiodo, and A. Sangiovanni-Vincentelli, "Trade-off evaluation in embedded system design via co-simulation," in *Proceedings of the ASP-DAC*, Jan. 1997.

[140] P. Paulin, C. Liem, T. May, and S. Sutarwala, "Flexware: A flexible firmware development environment for embedded systems," *Code Generators for Embedded Processors, P. Marwedel and G. Goossens, Eds. Amsterdam: Kluwer*, 1995.

[141] R. Perego and G. De Petris, "Minimizing network contention for mapping tasks onto massively parallel computers," in *Euromicro Workshop on Parallel and Distributed Processing*, pp. 210–218, Jan. 1995.

[142] A. Pnueli, "In transition from global to modular temporal reasoning about programs," *Logics and Models of Concurrent Systems*, pp. 123–144, 1989.

[143] H. Posadas, F. Herrera, P. Sanchez, E. Villar, and F. Blasco, "System-level performance analysis in SystemC," in *Design, Automation and Test in Europe Conference and Exhibition*, pp. 378–383, 2004.

[144] S. Prakash and A. C. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *Parallel Distributed Computing*, vol. 16, pp. 338351, 1992.

[145] D. Ragan, P. Sandborn, and P. Stoaks, "A detailed cost model for concurrent use with hardware/software co-design," in *39th Design Automation Conference*, pp. 269–274, 2002.

[146] R. Rajsuman, *System-on-a-Chip Design and Test*, Boston, MA: Kluwer, 2000.

[147] P. Ramanathan and J. Stankovic, "Scheduling algorithms and operating system support for real-time systems," *Proc. IEEE*, vol. 82, pp. 55–67, Jan. 1994.

[148] K. Ramani and R.L. Haggard, "A survey of techniques used in the synthesis of hardware from C/C++ as a part of hardware/software co-design," in *33rd Southeastern Symposium on System Theory*, pp. 301–304, Mar. 2001.

[149] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *14th Workshop Microprogramming*, p. 183198, Oct. 1981.

[150] Bertil Roslund and Per Andersson, "A flexible technique for OS-support in instruction level simulators," in *27th Simulation Symposium*, pp. 134–141, Apri. 1994.

[151] Jeffry T Russell and Margarida F Jacome, "Architecture-level performance evaluation of component-based embedded systems," in *Design Automation Conference*, pp. 396–401, Jun. 2003.

[152] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P.P. Pande, C. Grecu, and A. Ivanov, "System-on-chip: reuse and integration," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1050–1069, June 2006.

[153] T. Schubert, "High level formal verification of next-generation microprocessors," in *Design Automation Conference*, pp. 1–6, 2003.

[154] J. Seizovic, "Pipeline synchronization," in *IEEE International Symposim on Asynchronous Circuits and Systems*, pp. 87–96, 1994.

[155] D. Shapiro, *Globally-asynchronous, locally synchronous systems*, Ph.D. thesis, Computer Science Department, Standard University, Stanford, CA, 1984.

[156] R. Siegmund and D. Muller, "SystemCSV: an extension of SystemC for mixed multi-level communication modeling and interface-based system design," in DATE, 2001.

[157] I. Stoica, H. Abdel-Wahab, K. Jeffay, S.K. Baruah, J.E. Gehrke, and C.G. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *Real-Time Systems Symposium*, pp. 288–299, Dec. 1996.

[158] H. J. Stolberg, M. Berekovic, and P. Pirsch, "A platform-independent methodology for performance estimation of streaming media applications," in *Multimedia and Expo*, vol. 2, pp. 105–108, Aug. 2002.

[159] Wonyong Sung and Soonhoi Ha, "Optimized timed hardware software cosimulation without roll-back," in *Design, Automation and Test in Europe*, pp. 945–946, Feb. 1998.

[160] S. Swan, "SystemC transaction level models and RTL verification," in *43rd ACM/IEEE Design Automation Conference*, pp. 90–92, Jul. 2006.

[161] D. Thomas, J. Adams, and H. Schmitt, "A model and methodology for hardware-software co-design," *IEEE Design and Test*, vol. 10, no. 3, pp. 6–15, Sept. 1993.

[162] R. Tijdeman, "The chairmain assignment problem," *Discrete Mathematics*, vol. 32, pp. 323–330, 1980.

[163] Kyoko Ueda, Keishi Sakanushi, Yoshinori Takeuchi, and Masaharu Imai, "Architecture-level performance estimation for IP-based embedded systems," in *Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, pp. 1002–1007, Feb. 2004.

[164] F. Vahid, J. Gong, and D. Gajski, "A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning," in *Proc. EURODAC*, pp. 214–219, 1994.

[165] Catherine Lingxia Wang, Bo Yao, Yang Yang, and Zhengyong Zhu, "A survey of embedded operating system," http://www.cs.ucsd.edu/classes/fa01/cse221/projects/group2.pdf.

[166] Duen-Jeng Wang and Yu Hen Hu, "Fully static multiprocessor realization for real-time recursive DSP algorithms," in *International Conference on Application-Specific Array Processors*, pp. 664–678, Aug. 1992.

[167] Shige Wang, Sharath Kodase, Kang G. Shin, and Daniel L. Kiskis, "Measurement of OS services and its application to performance modeling and analysis of integrated embedded software," in *Real-Time and Embedded Technology and Applications Symposium*, pp. 113–122, 2002.

[168] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, Apri. 2003.

[169] Wooseung Yang, Moo-Kyeong Chung, and Chong-Min Kyung, "Current status and challenges of SoC verification for embedded system market," in *IEEE International conference on* SoC, pp. 213–216, Sept. 2003.

[170] Mitsuhiro Yasuida, Barry Shackleford, and Fumio Suzuki, "A top-down hardware/software co-simulation method for embedded systems based upon a component logical bus architecture," in *ASP-DAC*, pp. 169–175, Feb. 1998.

[171] T.-Y. Yen and W. Wolf, "Communicating synthesis for distributed embedded systems," in *ICCAD*, pp. 288–294, 1995.

[172] Y. Yi, D. Kim, and S. Ha, "Virtual synchronization technique with OS modeling for fast and time-accurate cosimulation," in *Hardware/Software Codesign and System Synthesis*, pp. 1–6, 2003.

[173] Sungjoo Yoo, *Performance Improvement of* HW/SW *Cosimulation Based on Synchronization Overhead Reduction*, Ph.D. thesis, Seoul National University, Korea, Feb. 2000.

[174] S. Yoo and K. Choi, "Synchronization overhead reduction in timed cosimulation," in *International High Level Design Validation and Test Workshop*, pp. 157–164, Nov. 1997.

[175] S. Yoo and A. A. Jerraya, "Hardware/software cosimulation from interface perspective," IEE *Proceedings-Computers and Digital Techniques*, vol. 152, no. 3, pp. 369–379, May 2005.

[176] Sungjoo Yoo, Babriela Nicolescu, Lovic Gauthier, and Ahmed A. Jerraya, "Automatic generation of fast timed simulation models for operating systems

in SoC design," in *Design, Automation and Test in Europe Conference and Exhibition*, pp. 620–627, 2002.

[177] Sungjoo Yoo, Gabriela Nicolescu, Damien Lyonnard, Amer Baghdadi, and Ahmed A. Jerraya, "A generic wrapper architecture for multi-processor SoC cosimulation and design," in *9th International Hardware/Software Codesign Symposium*, pp. 195–200, Apri. 2001.

[178] H. Yu, A. Gerstlauer, and D. Gajski, "RTOS scheduling in transaction level models," in *International Conference on Hardware/software Codesign and System Synthesis*, pp. 31–36, Oct. 2003.

[179] V. D. Zivkovic, E. Deprettere, P. van der Wolfa, and E. Kock, "From high level application specification to system-level architecture definition: Exploration, design and complitation," in *International Workshop on Compilers for Parallel Computers*, pp. 39–49, Jan. 2003.

[180] V. Zivojnovic and H. Meyr, "Compiled HW/SW co-simulation," in *33rd Design Automation Conference*, pp. 690–695, June 1996.

[181] V. Zivojnovit, S. Pees, and H. Meyr, "LISA: Machine description language and generic machine model for HW/SW co-design," in *International Workshop on VLSI Signal Processing*, 1996.

Vita

Zhengting He received his B.S.E.E. from Tsinghua University, Beijing, China in 2000 and M.S.E.E. from the The University of Texas at Austin in 2002. From May 2002 to August 2003, Zhengting worked at Texas Instruments in Houston, Texas for several terms as a co-op. From May 2004, he joined Texas Instruments in Houston, Texas as a full-time employee. Zhengting has been a member of the Institute of Electrical and Electronics Engineers (IEEE) since 2001.

Permanent Address: 7223 Sierra Night Dr.
Richmond, TX, 77469, U.S.A.

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.